Python Frequently Asked Questions

發🄵 *3.9.7*

Guido van Rossum
and the Python development team

11 月 05, 2021

Python Software Foundation
Email: docs@python.org

Contents

CHAPTER 1

常見 Python 問答集

1.1 常見資訊

1.1.1 什🄵是 Python?

Python 是一种解释型、交互式、面向对象的编程语言。它包含了模块、异常、动态类型、高层级动态数据类型以及类等特性。在面向对象编程以外它还支持多种编程范式，例如过程式和函数式编程等。Python 结合了超强的功能和极清晰的语法。它带有许多系统调用和库以及多种窗口系统的接口，并且能用 C 或 C++ 来进行扩展。它还可用作需要可编程接口的应用程序的扩展语言。最后，Python 非常易于移植：它可以在包括 Linux 和 macOS 在内的许多 Unix 变种以及 Windows 上运行。

要了解更多详情，请先查看 tutorial-index。Python 新手指南 提供了学习 Python 的其他入门教程及资源的链接。

1.1.2 什🄵是 Python 軟體基金會?

Python 軟體基金會是一個獨立非營利性組織，🄵且擁有 Python 版本 2.1 與更新的版本版權。Python 軟體基金會的任務在於精進相關於 Python 程式語言撰寫於開放原始碼技術，而且宣傳使用 Python 。Python 軟體基金會網址:https://www.python.org/psf/

在美國捐款給 Python 軟體基金會是免🄵的，如果你使用 Python 而且發現很好用，請貢獻捐款到 Python 軟體基金會捐款頁面 <https://www.python.org/psf/donations/>

1.1.3 當使用 Python 時有任何版權限制嗎?

你可以任意使用源码，只要你保留版权信息并在你基于 Python 的产品文档中显示该版权信息。如果你遵守此版权规则，就可以将 Python 用于商业领域，以源码或二进制码的形式（不论是否经过修改）销售 Python 的副本，或是以某种形式包含了 Python 的产品。当然，我们仍然希望获知所有对 Python 的商业使用。

請看 Python 軟體基金會的授權頁面 <https://www.python.org/psf/license/> 有更完整的授權🄵明

Python 的徽标是注册商标，在某些情况下需要获得允许方可使用。请参阅 商标使用政策 了解详情。

1.1.4 F什F Python 被創造出來

Guido van Rossum 寫下這篇" 非常長" 的簡述F明 Python 的由來

> 我在 CWI 的 ABC 部门时在实现解释型语言方面积累了丰富经验，通过与这个部门成员的协同工作，我学到了大量有关语言设计的知识。这是许多 Python 特性的最初来源，包括使用缩进来组织语句以及包含非常高层级的数据结构（虽然在 Python 中具体的实现细节完全不同）。
>
> 我对 ABC 语言有过许多抱怨，但同时也很喜欢它的许多特性。没有可能通过扩展 ABC 语言（或它的实现）来弥补我的不满——实际上缺乏可扩展性就是它最大的问题之一。我也有一些使用 Modula-2+ 的经验，并曾与 Modula-3 的设计者进行交流，还阅读了 Modula-3 的报告。Modula-3 是 Python 中异常机制所用语法和语义，以及其他一些语言特性的最初来源。
>
> 我还曾在 CWI 的 Amoeba 分布式操作系统部门工作。当时我们需要有一种比编写 C 程序或 Bash 脚本更好的方式来进行系统管理，因为 Amoeba 有它自己的系统调用接口，并且无法方便地通过 Bash 来访问。我在 Amoeba 中处理错误的经验令我深刻地意识到异常处理在编程语言特性当中的重要地位。
>
> 我发现，某种具有 ABC 式的语法而又能访问 Amoeba 系统调用的脚本语言将可满足需求。我意识到编写一种 Amoeba 专属的语言是愚蠢的，所以我决定编写一种具有全面可扩展性的语言。
>
> 在 1989 年的圣诞假期中，我手头的时间非常充裕，因此我决定开始尝试一下。在接下来的一年里，虽然我仍然主要用我的业余时间来做这件事，但 Python 在 Amoeba 项目中的使用获得了很大的成功，来自同事的反馈让我得以增加了许多早期的改进。
>
> 到 1991 年 2 月，经过一年多的开发，我决定将其发布到 USENET。之后的事情就都可以在 `Misc/HISTORY` 文件里面看了。

1.1.5 什F是 Python 擅長的事情

Python 是高階語言及一般任何用途都可以使用的語言，可以用來解F不同的問題

该语言附带一个庞大的标准库，涵盖了字符串处理（正则表达式，Unicode，比较文件间的差异等），因特网协议（HTTP，FTP，SMTP，XML-RPC，POP，IMAP，CGI 编程等），软件工程（单元测试，日志记录，性能分析，Python 代码解析等），以及操作系统接口（系统调用，文件系统，TCP/IP 套接字等）。请查看 library-index 的目录以了解所有可用的内容。此外还可以获取到各种各样的第三方扩展。请访问 Python 包索引 来查找你感兴趣的软件包。

1.1.6 Python 版本的编号形式是怎样的？

Python 版本的编号形式是 A.B.C 或 A.B。A 称为大版本号——它仅在对语言特性进行非常重大改变时才会递增。B 称为小版本号，它会在语言特性发生较小改变时递增。C 称为微版本号——它会在每次发布问题修正时递增。请参阅 **PEP 6** 了解有关问题修正发布版的详情。

发布版本并非都是问题修正版本。在新的主要发布版本开发过程中，还会发布一系列的开发版，它们以 alpha (a), beta (b) 或 release candidate (rc) 来标示。其中 alpha 版是早期发布的测试版，它的接口并未最终确定；在两个 alpha 发布版本间出现接口的改变并不意外。而 beta 版更为稳定，它会保留现有的接口，但也可能增加新的模块，release candidate 版则会保持冻结状态不会再进行改变，除非有重大问题需要修正。

以上 alpha, beta 和 release candidate 版本会附加一个后缀。用于 alpha 版本的后缀是带有一个小数字 N 的 "aN"，beta 版本的后缀是带有一个小数字 N 的 "bN"，而 release candidate 版本的后缀是带有一个小数字 N 的 "rcN"。换句话说，所有标记为 2.0aN 的版本都早于标记为 2.0bN 的版本，后者又都早于标记为 2.0rcN 的版本，而它们全都早于 2.0。

你还可能看到带有 "+" 后缀的版本号，例如 "2.2+"。这表示未发布版本，直接基于 CPython 开发代码仓库构建。在实际操作中，当一个小版本最终发布后，未发布版本号会递增到下一个小版本号，成为 "a0" 版本，例如 "2.4a0"。

另请参阅 `sys.version`, `sys.hexversion` 以及 `sys.version_info` 的文档。

1.1.7 我如何拿到 Python 的原始碼

最新的 Python 发布版源代码总能从 python.org 获取，下载页链接为 https://www.python.org/downloads/。最新的开发版源代码可以在 https://github.com/python/cpython/ 获取。

发布版源代码是一个以 gzip 压缩的 tar 文件，其中包含完整的 C 源代码、Sphinx 格式的文档、Python 库模块、示例程序以及一些有用的自由分发软件。该源代码将可在大多数 UNIX 类平台上直接编译并运行。

请参阅 Python 开发者指南的初步上手部分 了解有关获取源代码并进行编译的更多信息。

1.1.8 如何取得 Python 的相關文件

当前的 Python 稳定版本的标准文档可在 https://docs.python.org/3/ 查看。也可在 https://docs.python.org/3/download.html 获取 PDF、纯文本以及可下载的 HTML 版本。

文档以 reStructuredText 格式撰写，并使用 Sphinx 文档工具 生成。文档的 reStructuredText 源文件是 Python 源代码发布版的一部分。

1.1.9 我從來🄵寫過程式，有🄵有 Python 的教學

有许多可选择的教程和书籍。标准文档中也包含有 tutorial-index。

请参阅 新手指南 以获取针对 Python 编程初学者的信息，包括教程的清单。

1.1.10 有🄵有新手的群組或是郵件群組討論 Python

有一个新闻组 `comp.lang.python` 和一个邮件列表 python-list。新闻组和邮件列表是彼此互通的——如果你可以阅读新闻就不必再订阅邮件列表。`comp.lang.python` 的流量很大，每天会收到数以百计的发帖，Usenet 使用者通常更擅长处理这样大的流量。

有关新软件发布和活动的公告可以在 comp.lang.python.announce 中找到，这是个严格管理的低流量列表，每天会收到五个左右的发帖。可以在 Python 公告邮件列表 页面进行订阅。

有关其他邮件列表和新闻组的更多信息可以在 https://www.python.org/community/lists/ 找到。

1.1.11 我应如何获取 Python 的公开测试版本？

可以从 https://www.python.org/downloads/ 下载 alpha 和 beta 发布版。所有发布版都会在 comp.lang.python 和 comp.lang.python.announce 新闻组以及 Python 主页 https://www.python.org/ 上进行公告；并会推送到 RSS 新闻源。

你还可以通过 Git 访问 Python 的开发版。请参阅 Python 开发者指南 了解详情。

1.1.12 我应如何为 Python 提交错误报告和补丁？

要报告错误或提交补丁，请使用安装于 https://bugs.python.org/ 上的 Roundup。

你必须拥有一个 Roundup 账号才能报告错误；这样我们就可以在有后续问题时与你联系。这也使得 Roundup 能在我们处理所报告的错误时向你发送更新消息。如果你之前使用过 SourceForge 向 Python 报告错误，你可以通过 Roundup 的 密码重置操作 来获取你的 Roundup 密码。

有关 Python 开发流程的更多信息，请参阅 Python 开发者指南。

1.1.13 是否有任何公开发表的 Python 相关文章可以供我参考引用？

可能作为参考文献的最好方式还是引用你喜欢的 Python 相关书籍。

最早討論 Python 的文章在 1991 年，但現在來看已經有點過時

> Guido van Rossum 与 Jelke de Boer, " 使用 Python 编程语言交互式地测试远程服务器", CWI 季刊, 第 4 卷, 第 4 期 (1991 年 12 月), 阿姆斯特丹, 第 283--303 页。

1.1.14 有🄵有關於 Python 的書

是的已經有很多書出版，可以參考這個連結的參考書目 https://wiki.python.org/moin/PythonBooks

你也可以上網搜尋網路書店關鍵字🄵"Python"，但不要使用"Monty Python" 當作關鍵字。或者可以搜尋"Python" 和" 語言"

1.1.15 www.python.org 這個非營利組織位於哪🄵

Python 项目的基础架构分布于世界各地并由 Python 基础架构团队负责管理。详情请访问 这里。

1.1.16 🄵什🄵要取名🄵 Python

在着手编写 Python 实现的时候，Guido van Rossum 同时还阅读了刚出版的 "Monty Python 的飞行马戏团" 剧本，这是一部自 1970 年代开始播出的 BBC 系列喜剧。Van Rossum 觉得他需要选择一个简短、独特而又略显神秘的名字，于是他决定将这个新语言命名为 Python。

1.1.17 我需要喜歡蒙提 · 派森的飛行馬戲團這個節目嗎

不需要，但它有幫助:)

1.2 Python 在真實世界

1.2.1 Python 穩定性如何

非常穩定。自從 1991 年開始大約每隔 6 到 18 個月會釋出更新版，而且看起來會繼續更新下去。從 3.9 開始，Python 每隔 12 個月會釋出一個主要發行版本 (**PEP 602**)。

开发者也会推出旧版本的“问题修正”发布版，因此现有发布版的稳定性还会逐步提升。问题修正发布版会以版本号第三部分的数字来标示（例如 3.5.3, 3.6.2），用于稳定性的管理；只有对已知问题的修正会包含在问题修正发布版中，同一系列的问题修正发布版中的接口确定将会始终保持一致。

最后的稳定版本总是可以在 Python 下载页 中找到。有两个生产环境可用的 Python 版本：2.x 和 3.x。推荐的版本是 3.x，大多数被广泛使用的库都支持它。虽然 2.x 也仍然被广泛使用，但是 它已经停止维护。

1.2.2 有多少人使用 Python

大約有超過一百萬個使用者，但實際上有多少人是很難准確的估算

Python 可以免费下载，因此并不存在销量数据，此外它也可以从许多不同网站获取，并且包含于许多 Linux 发行版之中，因此下载量统计同样无法完全说明问题。

comp.lang.python 新闻组非常活跃，但不是所有 Python 用户都会在新闻组发帖，许多人甚至不会阅读新闻组。

1.2.3 有🄵有任何重要的案子使用 Python 完成開發

请访问 https://www.python.org/about/success 查看使用了 Python 的项目列表。阅览 历次 Python 会议 的日程纪要可以看到许多不同公司和组织所做的贡献。

高水准的 Python 项目包括 Mailman 邮件列表管理器 和 Zope 应用服务器。多个 Linux 发行版，其中最著名的有 Red Hat 均已使用 Python 来编写部分或全部的安装程序和系统管理软件。在内部使用 Python 的大公司包括了 Google, Yahoo 以及 Lucasfilm 等。

1.2.4 對於程式開發者 Python 對於未來有什🄵期待

请访问 https://www.python.org/dev/peps/ 查看 Python 增强提议（PEP）。PEP 是为 Python 加入某种新特性的提议进行描述的设计文档，其中会提供简明的技术规格说明与基本原理。可以查找标题为 “Python X.Y Release Schedule” 的 PEP，其中 X.Y 是某个尚未公开发布的版本。

新版本的开发会在 python-dev 邮件列表 中进行讨论。

1.2.5 提议对 Python 加入不兼容的更改是否合理？

通常来说是不合理的。世界上已存在的 Python 代码数以亿计，因此，任何对该语言的更改即便仅会使得现有程序中极少的一部分失效也是难以令人接受的。就算你可以提供一个转换程序，也仍然存在需要更新全部文档的问题；另外还有大量已出版的 Python 书籍，我们不希望让它们在一瞬间全部变成废纸。

如果必须更改某个特性，则应该提供渐进式的升级路径。**PEP 5** 描述了引入向后不兼容的更改所需遵循的流程，以尽可能减少对用户的干扰。

1.2.6 Python 對於入門的程式設計者而言是否🄵好的程式語言

是的

从过程式、静态类型的编程语言例如 Pascal, C 或者 C++ 以及 Java 的某一子集开始引导学生入门仍然是常见的做法。但以 Python 作为第一种编程语言进行学习对学生可能更有利。Python 具有非常简单和一致的语法和庞大的标准库，而且最重要的是，在编程入门教学中使用 Python 可以让学生专注于更重要的编程技能，例如问题分解与数据类型设计。使用 Python，可以快速向学生介绍基本概念例如循环与过程等。他们甚至有可能在第一次课里就开始接触用户自定义对象。

对于之前从未接触过编程的学生来说，使用静态类型语言会感觉不够自然。这会给学生带来必须掌握的额外复杂性，并减慢教学的进度。学生需要尝试像计算机一样思考，分解问题，设计一致的接口并封装数据。虽然从长远来看，学习和使用一种静态类型语言是很重要的，但这并不是最适宜在学生的第一次编程课上就进行探讨的主题。

还有许多其他方面的特点使得 Python 成为很好的入门语言。像 Java 一样，Python 拥有一个庞大的标准库，因此可以在课程非常早期的阶段就给学生布置一些 实用的编程项目。编程作业不必仅限于标准四则运算和账目检查程序。通过使用标准库，学生可以在学习编程基础知识的同时开发真正的应用，从而获得更大的满足感。使用标准库还能使学生了解代码重用的概念。而像 PyGame 这样的第三方模块同样有助于扩大学生的接触领域。

Python 的解释器使学生能够在编程时测试语言特性。他们可以在一个窗口中输入程序源代码的同时开启一个解释器运行窗口。如果他们不记得列表有哪些方法，他们这以这样做:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

通过使用解释器，学生编写程序时参考文档总是能伴随在他们身边。

Python 还拥有很好的 IDE。IDLE 是一个跨平台的 Python IDE，它基于 Tkinter 库，使用 Python 语言编写。PythonWin 是一个 Windows 专属的 IDE。Emacs 用户将高兴地了解到 Emacs 具有非常好的 Python 模式。所有这些编程环境都提供语法高亮，自动缩进以及在编写代码时使用交互式解释器等功能。请访问 Python wiki 查看 Python 编程环境的完整列表。

如果你想要讨论 Python 在教育中的使用，你可能会有兴趣加入 edu-sig 邮件列表。

CHAPTER 2

程式開發常見問答集

2.1 常見問題

2.1.1 是否有可以使用在程式碼階段, 具有中斷點, 步驟執行等功能的除錯器？

有的

以下介绍了一些 Python 的调试器，用内置函数 `breakpoint()` 即可切入这些调试器中。

pdb 模块是一个简单但是够用的控制台模式 Python 调试器。它是标准 Python 库的一部分，并且 已收录于 *库参考手册*。你也可以通过使用 pdb 代码作为样例来编写你自己的调试器。

作为标准 Python 发行版附带组件的 IDLE 交互式环境（通常位于 Tools/scripts/idle）中包含一个图形化的调试器。

PythonWin 是一种 Python IDE，其中包含了一个基于 pdb 的 GUI 调试器。PythonWin 的调试器会为断点着色，并提供了相当多的超酷特性，例如调试非 PythonWin 程序等。PythonWin 是 pywin32 项目的组成部分，也是 ActivePython 发行版的组成部分。

Eric 是一个基于 PyQt 和 Scintilla 编辑组件构建的 IDE。

trepan3k 是一个类似 gdb 的调试器。

Visual Studio Code 是包含了调试工具的 IDE，并集成了版本控制软件。

有數個商業化 Python 整合化開發工具包含圖形除錯功能。這些包含：

- Wing IDE
- Komodo IDE
- PyCharm

2.1.2 有🄵有工具能🄵幫忙找 bug 或執行🄵態分析？

有的

Pylint 和 Pyflakes 可进行基本检查来帮助你尽早捕捉漏洞。

静态类型检查器，例如 Mypy 、Pyre 和 Pytype 可以检查 Python 源代码中的类型提示。

2.1.3 如何由 Python 脚本创建能独立运行的二进制程序？

如果只是想要一个独立的程序，以便用户不必预先安装 Python 即可下载和运行它，则不需要将 Python 编译成 C 代码。有许多工具可以检测程序所需的模块，并将这些模块与 Python 二进制程序捆绑在一起生成单个可执行文件。

一种方案是使用 freeze 工具，它在 Python 源码目录的 `Tools/freeze` 中提供。它能把 Python 字节码转换为 C 数组；一个 C 编译器可以将所有模块嵌入到一个新程序中，然后将其与标准 Python 模块链接。

它的工作原理是递归扫描源代码，获取两种格式的 import 语句，并在标准 Python 路径和源码目录（用于内置模块）检索这些模块。然后，把这些模块的 Python 字节码转换为 C 代码（可以利用 marshal 模块转换为代码对象的数组初始化器），并创建一个定制的配置文件，该文件仅包含程序实际用到的内置模块。然后，编译生成的 C 代码并将其与 Python 解释器的其余部分链接，形成一个自给自足的二进制文件，其功能与 Python 脚本代码完全相同。

下列包可以用于帮助创建控制台和 GUI 的可执行文件:

- Nuitka （跨平台）
- PyInstaller （跨平台）
- PyOxidizer （跨平台）
- cx_Freeze （跨平台）
- py2app （仅限 macOS）
- py2exe （僅限 Windows）

2.1.4 是否有 Python 编码标准或风格指南？

有的。标准库模块所要求的编码风格记录于 **PEP 8** 之中。

2.2 语言核心内容

2.2.1 🄵什🄵當變數有值時我得到錯誤訊息 UnboundLocalError

因为在函数内部某处添加了一条赋值语句，导致之前正常工作的代码报出 UnboundLocalError 错误，这可能是有点令人惊讶。

這段程式碼:

```
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

可以執行, 但是這段程式:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

導致 UnboundLocalError

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

原因就是，当对某作用域内的变量进行赋值时，该变量将成为该作用域内的局部变量，并覆盖外部作用域中的同名变量。由于 foo 的最后一条语句为 x 分配了一个新值，编译器会将其识别为局部变量。因此，前面的 `print(x)` 试图输出未初始化的局部变量，就会引发错误。

在上面的示例中，可以将外部作用域的变量声明为全局变量以便访问：

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

与类和实例变量貌似但不一样，其实以上是在修改外部作用域的变量值，为了提示这一点，这里需要显式声明一下。

```
>>> print(x)
11
```

你可以使用 `nonlocal` 关键字在嵌套作用域中执行类似的操作：

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
10
11
```

2.2.2 Python 的區域變數和全域變數有什🄵規則？

函数内部只作引用的 Python 变量隐式视为全局变量。如果在函数内部任何位置为变量赋值，则除非明确声明为全局变量，否则均将其视为局部变量。

起初尽管有点令人惊讶，不过考虑片刻即可释然。一方面，已分配的变量要求加上 `global` 可以防止意外的副作用发生。另一方面，如果所有全局引用都要加上 `global` ，那处处都得用上 `global` 了。那么每次对内置函数或导入模块中的组件进行引用时，都得声明为全局变量。这种杂乱会破坏 `global` 声明用于警示副作用的有效性。

2.2.3 为什么在循环中定义的参数各异的 lambda 都返回相同的结果？

假设用 for 循环来定义几个取值各异的 lambda（即便是普通函数也一样）：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

以上会得到一个包含 5 个 lambda 函数的列表，这些函数将计算 x**2。大家或许期望，调用这些函数会分别返回 0 、 1 、 4 、 9 和 16。然而，真的试过就会发现，他们都会返回 16：

```
>>> squares[2]()
16
>>> squares[4]()
16
```

这是因为 x 不是 lambda 函数的内部变量，而是定义于外部作用域中的，并且 x 是在调用 lambda 时访问的——而不是在定义时访问。循环结束时 x 的值是 4 ，所以此时所有的函数都将返回 4**2 ，即 16 。通过改变 x 的值并查看 lambda 的结果变化，也可以验证这一点。

```
>>> x = 8
>>> squares[2]()
64
```

为了避免发生上述情况，需要将值保存在 lambda 局部变量，以使其不依赖于全局 x 的值：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

以上 n=x 创建了一个新的 lambda 本地变量 n，并在定义 lambda 时计算其值，使其与循环当前时点的 x 值相同。这意味着 n 的值在第 1 个 lambda 中为 0 ，在第 2 个 lambda 中为 1 ，在第 3 个中为 2，依此类推。因此现在每个 lambda 都会返回正确结果：

```
>>> squares[2]()
4
>>> squares[4]()
16
```

请注意，上述表现并不是 lambda 所特有的，常规的函数也同样适用。

2.2.4 如何跨模块共享全局变量？

在单个程序中跨模块共享信息的规范方法是创建一个特殊模块（通常称为 config 或 cfg）。只需在应用程序的所有模块中导入该 config 模块；然后该模块就可当作全局名称使用了。因为每个模块只有一个实例，所以对该模块对象所做的任何更改将会在所有地方得以体现。例如：

config.py：

```
x = 0   # Default value of the 'x' configuration setting
```

mod.py：

```
import config
config.x = 1
```

main.py：

```
import config
import mod
print(config.x)
```

请注意，出于同样的原因，采用模块也是实现单例设计模式的基础。

2.2.5 导入模块的“最佳实践”是什么？

通常请勿使用 `from modulename import *` 。因为这会扰乱 importer 的命名空间，且会造成未定义名称更难以被 Linter 检查出来。

请在代码文件的首部就导入模块。这样代码所需的模块就一目了然了，也不用考虑模块名是否在作用域内的问题。每行导入一个模块则增删起来会比较容易，每行导入多个模块则更节省屏幕空间。

按如下顺序导入模块就是一种好做法：

1. 标准库模块——比如：`sys` 、`os` 、`getopt` 、`re` 等。
2. 第三方库模块（安装于 Python site-packages 目录中的内容）——如 mx.DateTime、ZODB、PIL.Image 等。
3. 本地开发的模块

为了避免循环导入引发的问题，有时需要将模块导入语句移入函数或类的内部。Gordon McMillan 的说法如下：

> 当两个模块都采用”import <module>” 的导入形式时，循环导入是没有问题的。但如果第 2 个模块想从第 1 个模块中取出一个名称（”from module import name”）并且导入处于代码的最顶层，那导入就会失败。原因是第 1 个模块中的名称还不可用，这时第 1 个模块正忙于导入第 2 个模块呢。

如果只是在一个函数中用到第 2 个模块，那这时将导入语句移入该函数内部即可。当调用到导入语句时，第 1 个模块将已经完成初始化，第 2 个模块就可以进行导入了。

如果某些模块是平台相关的，可能还需要把导入语句移出最顶级代码。这种情况下，甚至有可能无法导入文件首部的所有模块。于是在对应的平台相关代码中导入正确的模块，就是一种不错的选择。

只有为了避免循环导入问题，或有必要减少模块初始化时间时，才把导入语句移入类似函数定义内部的局部作用域。如果根据程序的执行方式，许多导入操作不是必需的，那么这种技术尤其有用。如果模块仅在某个函数中用到，可能还要将导入操作移入该函数内部。请注意，因为模块有一次初始化过程，所以第一次加载模块的代价可能会比较高，但多次加载几乎没有什么花费，代价只是进行几次字典检索而已。即使模块名超出了作用域，模块在 `sys.modules` 中也是可用的。

2.2.6 为什么对象之间会共享默认值？

新手程序员常常中招这类 Bug。请看以下函数：

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

第一次调用此函数时，`mydict` 中只有一个数据项。第二次调用 `mydict` 则会包含两个数据项，因为 `foo()` 开始执行时，`mydict` 中已经带有一个数据项了。

大家往往希望，函数调用会为默认值创建新的对象。但事实并非如此。默认值只会在函数定义时创建一次。如果对象发生改变，就如上例中的字典那样，则后续调用该函数时将会引用这个改动的对象。

按照定义，不可变对象改动起来是安全的，诸如数字、字符串、元组和 `None` 之类。而可变对象的改动则可能引起困惑，例如字典、列表和类实例等。

因此，不把可变对象用作默认值是一种良好的编程做法。而应采用 `None` 作为默认值，然后在函数中检查参数是否为 `None` 并新建列表、字典或其他对象。例如，代码不应如下所示：

```
def foo(mydict={}):
    ...
```

而应这么写：

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

参数默认值的特性有时会很有用处。如果有个函数的计算过程会比较耗时，有一种常见技巧是将每次函数调用的参数和结果缓存起来，并在同样的值被再次请求时返回缓存的值。这种技巧被称为“memoize”，实现代码可如下所示：

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

也可以不用参数默认值来实现，而是采用全局的字典变量；这取决于个人偏好。

2.2.7 如何将可选参数或关键字参数从一个函数传递到另一个函数？

请利用函数参数列表中的标识符 * 和 ** 归集实参；结果会是元组形式的位置实参和字典形式的关键字实参。然后就可利用 * 和 ** 在调用其他函数时传入这些实参：

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 形参和实参之间有什么区别？

形参 是指出现在函数定义中的名称，而*实参* 则是在调用函数时实际传入的值。形参定义了一个函数能接受何种类型的实参。例如，对于以下函数定义：

```
def func(foo, bar=None, **kwargs):
    pass
```

foo 、*bar* 和 *kwargs* 是 `func` 的形参。不过在调用 `func` 时，例如：

```
func(42, bar=314, extra=somevar)
```

`42` 、`314` 和 `somevar` 则是实参。

2.2.9 为什么修改列表'y' 也会更改列表'x'？

如果代码编写如下：

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

或许大家很想知道，为什么在 y 中添加一个元素时，x 也会改变。

产生这种结果有两个因素：

1) 变量只是指向对象的一个名称。执行 `y = x` 并不会创建列表的副本——而只是创建了一个新变量 y，并指向 x 所指的同一对象。这就意味着只存在一个列表对象，x 和 y 都是对它的引用。

2) 列表属于*mutable* 对象，这意味着它的内容是可以修改的。

在调用 `append()` 之后，该可变对象的内容由 `[]` 变为 `[10]`。由于 x 和 y 这两个变量引用了同一对象，因此用其中任意一个名称所访问到的都是修改后的值 `[10]`。

如果把赋给 x 的对象换成一个不可变对象：

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

可见这时 x 和 y 就不再相等了。因为整数是*immutable* 对象，在执行 `x = x + 1` 时，并不会修改整数对象 5，给它加上 1；而是创建了一个新的对象（整数对象 6）并将其赋给 x（也就是改变了 x 所指向的对象）。在赋值完成后，就有了两个对象（整数对象 6 和 5）和分别指向他俩的两个变量（x 现在指向 6 而 y 仍然指向 5）。

某些操作（例如 `y.append(10)` 和 `y.sort()`）会直接修改原对象，而看上去相似的另一些操作（例如 `y = y + [10]` 和 `sorted(y)`）则会创建新的对象。通常在 Python 中（以及所有标准库），直接修改原对象的方法将会返回 `None`，以助避免混淆这两种不同类型的操作。因此如果误用了 `y.sort()` 并期望返回 y 的有序副本，则结果只会是 `None`，这可能就能让程序引发一条容易诊断的错误。

不过还存在一类操作，用不同的类型执行相同的操作有时会发生不同的行为：即增量赋值运算符。例如，`+=` 会修改列表，但不会修改元组或整数（`a_list += [1, 2, 3]` 与 `a_list.extend([1, 2, 3])` 同样都会改变 `a_list`，而 `some_tuple += (1, 2, 3)` 和 `some_int += 1` 则会创建新的对象）。

换而言之：

- 对于一个可变对象（`list`、`dict`、`set` 等等），可以利用某些特定的操作进行修改，所有引用它的变量都会反映出改动情况。
- 对于一个不可变对象（`str`、`int`、`tuple` 等），所有引用它的变量都会给出相同的值，但所有改变其值的操作都将返回一个新的对象。

如要知道两个变量是否指向同一个对象，可以利用 `is` 运算符或内置函数 `id()`。

2.2.10 如何编写带有输出参数的函数（按照引用调用）？

请记住，Python 中的实参是通过赋值传递的。由于赋值只是创建了对象的引用，所以调用方和被调用方的参数名都不存在别名，本质上也就不存在按引用调用的方式。通过以下几种方式，可以得到所需的效果。

1) 返回一个元组：

```
>>> def func1(a, b):
...     a = 'new-value'        # a and b are local names
...     b = b + 1              # assigned to new objects
...     return a, b            # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

这差不多是最明晰的解决方案了。

2) 使用全局变量。这不是线程安全的方案，不推荐使用。

3) 传递一个可变（即可原地修改的）对象：

```
>>> def func2(a):
...     a[0] = 'new-value'     # 'a' references a mutable list
...     a[1] = a[1] + 1        # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4) 传入一个接收可变对象的字典:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args is a mutable dictionary
...     args['b'] = args['b'] + 1   # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5) 或者把值用类实例封装起来：

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

没有什么理由要把问题搞得这么复杂。

最佳选择就是返回一个包含多个结果值的元组。

2.2.11 如何在 Python 中创建高阶函数？

有两种选择：嵌套作用域、可调用对象。假定需要定义 `linear(a,b)`，其返回结果是一个计算出 `a*x+b` 的函数 `f(x)`。采用嵌套作用域的方案如下：

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

或者可采用可调用对象：

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b
```

(下页继续)

(繼續上一頁)

```
    def __call__(self, x):
        return self.a * x + self.b
```

采用这两种方案时：

```
taxes = linear(0.3, 2)
```

都会得到一个可调用对象，可实现 `taxes(10e6) == 0.3 * 10e6 + 2` 。

可调用对象的方案有个缺点，就是速度稍慢且生成的代码略长。不过值得注意的是，同一组可调用对象能够通过继承来共享签名（类声明）：

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

对象可以为多个方法的运行状态进行封装：

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

以上 `inc()` 、`dec()` 和 `reset()` 的表现，就如同共享了同一计数变量一样。

2.2.12 如何复制 Python 对象？

一般情况下，用 `copy.copy()` 或 `copy.deepcopy()` 基本就可以了。并不是所有对象都支持复制，但多数是可以的。

某些对象可以用更简便的方法进行复制。比如字典对象就提供了 `copy()` 方法：

```
newdict = olddict.copy()
```

序列可以用切片操作进行复制：

```
new_l = l[:]
```

2.2.13 如何找到对象的方法或属性？

假定 x 是一个用户自定义类的实例，`dir(x)` 将返回一个按字母排序的名称列表，其中包含了实例的属性及由类定义的方法和属性。

2.2.14 如何用代码获取对象的名称？

一般而言这是无法实现的，因为对象并不存在真正的名称。赋值本质上是把某个名称绑定到某个值上；`def` 和 `class` 语句同样如此，只是值换成了某个可调用对象。比如以下代码：

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

可以不太严谨地说，上述类具有一个名称：即便它绑定了两个名称并通过名称 B 发起调用，可是创建出来的实例仍被视为是类 A 的实例。但无法说出实例的名称是 a 还是 b，因为这两个名称都被绑定到同一个值上了。

代码一般没有必要去“知晓”某个值的名称。通常这种需求预示着还是改变方案为好，除非真的是要编写内审程序。

在 comp.lang.python 中，Fredrik Lundh 在回答这样的问题时曾经给出过一个绝佳的类比：

> 这就像要知道家门口的那只猫的名字一样：猫（对象）自己不会说出它的名字，它根本就不在乎自己叫什么——所以唯一方法就是问一遍你所有的邻居（命名空间），这是不是他们家的猫（对象）……
>
> ……并且如果你发现它有很多名字或根本没有名字，那也不必惊讶！

2.2.15 逗号运算符的优先级是什么？

逗号不是 Python 的运算符。请看以下例子：

```
>>> "a" in "b", "a"
(False, 'a')
```

由于逗号不是运算符，而只是表达式之间的分隔符，因此上述代码就相当于：

```
("a" in "b"), "a"
```

而不是：

```
"a" in ("b", "a")
```

对于各种赋值运算符（= 、+= 等）来说同样如此。他们并不是真正的运算符，而只是赋值语句中的语法分隔符。

2.2.16 是否提供等价于 C 语言"?:" 三目运算符的东西？

有的。语法如下：

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

在 Python 2.5 引入上述语法之前，通常的做法是使用逻辑运算符：

```
[expression] and [on_true] or [on_false]
```

然而这种做法并不保险，因为当 *on_true* 为布尔值"假"时，结果将会出错。所以肯定还是采用 `... if ... else ...` 形式为妙。

2.2.17 是否可以用 Python 编写让人眼晕的单行程序?

可以。通常是在 `lambda` 中嵌套 `lambda` 来实现的。请参阅以下三个来自 Ulf Bartelt 的示例代码：

```
from functools import reduce

# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Mandelbrot set
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |_________ maximum of "iterations"
#         |          |________________ range on y axis
#         |___________________________ range on x axis
```

请不要在家里尝试，骚年！

2.2.18 函数形参列表中的斜杠（/）是什么意思？

函数参数列表中的斜杠表示在它之前的形参全都仅限位置形参。仅限位置形参没有可供外部使用的名称。在调用仅接受位置形参的函数时，实参只会根据位置映射到形参上。假定 `divmod()` 是一个仅接受位置形参的函数。它的帮助文档如下所示：

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

形参列表尾部的斜杠说明，两个形参都是仅限位置形参。因此，用关键字参数调用 `divmod()` 将会引发错误：

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 数字和字符串

2.3.1 如何给出十六进制和八进制整数？

要给出八进制数，需在八进制数值前面加上一个零和一个小写或大写字母"o" 作为前缀。例如，要将变量"a" 设为八进制的"10" （十进制的 8），写法如下：

```
>>> a = 0o10
>>> a
8
```

十六进制数也很简单。只要在十六进制数前面加上一个零和一个小写或大写的字母"x"。十六进制数中的字母可以为大写或小写。比如在 Python 解释器中输入：

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 为什么 -22 // 10 会返回 -3 ？

这主要是为了让 `i % j` 的正负与 `j` 一致，如果期望如此，且期望如下等式成立：

```
i == (i // j) * j + (i % j)
```

那么整除就必须返回向下取整的结果。C 语言同样要求保持这种一致性，于是编译器在截断 `i // j` 的结果时需要让 `i % j` 的正负与 `i` 一致。

对于 `i % j` 来说 `j` 为负值的应用场景实际上是非常少的。而 `j` 为正值的情况则非常多，并且实际上在所有情况下让 `i % j` 的结果为 `>= 0` 会更有用处。如果现在时间为 10 时，那么 200 小时前应是几时？`-190 % 12 == 2` 是有用处的；`-190 % 12 == -10` 则是会导致意外的漏洞。

2.3.3 How do I get int literal attribute instead of SyntaxError?

Trying to lookup an `int` literal attribute in the normal manner gives a syntax error because the period is seen as a decimal point:

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

The solution is to separate the literal from the period with either a space or parentheses.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

2.3.4 如何将字符串转换为数字？

对于整数，可使用内置的 `int()` 类型构造器，例如 `int('144') == 144`。类似地，可使用 `float()` 转换为浮点数，例如 `float('144') == 144.0`。

默认情况下，这些操作会将数字按十进制来解读，因此 `int('0144') == 144` 为真值，而 `int('0x144')` 会引发 `ValueError`。`int(string, base)` 接受第二个可选参数指定转换的基数，例如 `int( '0x144', 16) == 324`。如果指定基数为 0，则按 Python 规则解读数字：前缀'0o' 表示八进制，而'0x' 表示十六进制。

如果只是想把字符串转为数字，请不要使用内置函数 `eval()`。`eval()` 的速度慢很多且存在安全风险：别人可能会传入带有不良副作用的 Python 表达式。比如可能会传入 `__import__('os').system("rm -rf $HOME")`，这会把 home 目录给删了。

`eval()` 还有把数字解析为 Python 表达式的后果，因此如 `eval('09')` 将会导致语法错误，因为 Python 不允许十进制数带有前导'0'（'0' 除外）。

2.3.5 如何将数字转换为字符串？

比如要把数字 144 转换为字符串'144'，可使用内置类型构造器 `str()`。如果要表示为十六进制或八进制数格式，可使用内置函数 `hex()` 或 `oct()`。更复杂的格式化方法请参阅 f-strings 和 formatstrings 等章节，比如 `"{:04d}".format(144)` 会生成 `'0144'`，`"{:.3f}".format(1.0/3.0)` 则会生成 `'0.333'`。

2.3.6 如何修改字符串？

无法修改，因为字符串是不可变对象。在大多数情况下，只要将各个部分组合起来构造出一个新字符串即可。如果需要一个能原地修改 Unicode 数据的对象，可以试试 `io.StringIO` 对象或 `array` 模块：

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.7 如何使用字符串调用函数/方法？

有多种技巧可供选择。

- 最好的做法是采用一个字典，将字符串映射为函数。其主要优势就是字符串不必与函数名一样。这也是用来模拟 case 结构的主要技巧：

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- 利用内置函数 `getattr()`：

```
import foo
getattr(foo, 'bar')()
```

  请注意 `getattr()` 可用于任何对象，包括类、类实例、模块等等。

  标准库就多次使用了这个技巧，例如：

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- 用 `locals()` 解析出函数名：

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

2.3.8 是否有与 Perl 的 chomp() 等效的方法，用于从字符串中删除尾随换行符？

可以使用 `S.rstrip("\r\n")` 从字符串 `S` 的末尾删除所有的换行符，而不删除其他尾随空格。如果字符串 `S` 表示多行，且末尾有几个空行，则将删除所有空行的换行符：

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

由于通常只在一次读取一行文本时才需要这样做，所以使用 `S.rstrip()` 这种方式工作得很好。

2.3.9 是否有 scanf() 或 sscanf() 的等价函数？

没有。

如果要对简单的输入进行解析，最容易的做法通常是利用字符串对象的 `split()` 方法将一行按空白符分隔拆分为多个单词，然后用 `int()` 或 `float()` 将十进制数字符串转换为数字值。`split()` 支持可选的"sep" 形参，适用于分隔符不用空白符的情况。

如果要对更复杂的输入进行解析，那么正则表达式要比 C 语言的 `sscanf()` 更强大，也更合适。

2.3.10 'UnicodeDecodeError' 或'UnicodeEncodeError' 错误是什么意思？

见 unicode-howto

2.4 性能

2.4.1 我的程序太慢了。该如何加快速度？

总的来说，这是个棘手的问题。在进一步讨论之前，首先应该记住以下几件事：

- 不同的 Python 实现具有不同的性能特点。本 FAQ 着重解答的是*CPython*。
- 不同操作系统可能会有不同表现，尤其是涉及 I/O 和多线程时。
- 在尝试优化代码 之前，务必要先找出程序中的热点（请参阅 `profile` 模块）。
- 编写基准测试脚本，在寻求性能提升的过程中就能实现快速迭代（请参阅 `timeit` 模块）。
- 强烈建议首先要保证足够高的代码测试覆盖率（通过单元测试或其他技术），因为复杂的优化有可能会导致代码回退。

话虽如此，Python 代码的提速还是有很多技巧的。以下列出了一些普适性的原则，对于让性能达到可接受的水平会有很大帮助：

- 相较于试图对全部代码铺开做微观优化，优化算法（或换用更快的算法）可以产出更大的收益。
- 使用正确的数据结构。参考 bltin-types 和 `collections` 模块的文档。
- 如果标准库已为某些操作提供了基础函数，则可能（当然不能保证）比所有自编的函数都要快。对于用 C 语言编写的基础函数则更是如此，比如内置函数和一些扩展类型。例如，一定要用内置方法 `list.sort()` 或 `sorted()` 函数进行排序（某些高级用法的示例请参阅 sortinghowto ）。
- 抽象往往会造成中间层，并会迫使解释器执行更多的操作。如果抽象出来的中间层级太多，工作量超过了要完成的有效任务，那么程序就会被拖慢。应该避免过度的抽象，而且往往也会对可读性产生不利影响，特别是当函数或方法比较小的时候。

如果你已经达到纯 Python 允许的限制，那么有一些工具可以让你走得更远。例如，Cython 可以将稍微修改的 Python 代码版本编译为 C 扩展，并且可以在许多不同的平台上使用。Cython 可以利用编译（和可选的类型注释）来使代码明显快于解释运行时的速度。如果您对 C 编程技能有信心，也可以自己 编写 C 扩展模块。

也参考：

专门介绍 性能提示 的 wiki 页面。

2.4.2 将多个字符串连接在一起的最有效方法是什么？

str 和 bytes 对象是不可变的，因此连接多个字符串的效率会很低，因为每次连接都会创建一个新的对象。一般情况下，总耗时与字符串总长是二次方的关系。

如果要连接多个 str 对象，通常推荐的方案是先全部放入列表，最后再调用 str.join()：

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

（还有一种合理高效的习惯做法，就是利用 io.StringIO）

如果要连接多个 bytes 对象，推荐做法是用 bytearray 对象的原地连接操作（+= 运算符）追加数据：

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 序列（元组/列表）

2.5.1 如何在元组和列表之间进行转换？

类型构造器 tuple(seq) 可将任意序列（实际上是任意可迭代对象）转换为数据项和顺序均不变的元组。

例如，tuple([1, 2, 3]) 会生成 (1, 2, 3)，tuple('abc') 则会生成 ('a', 'b', 'c')。如果参数就是元组，则不会创建副本而是返回同一对象，因此如果无法确定某个对象是否为元组时，直接调用 tuple() 也没什么代价。

类型构造器 list(seq) 可将任意序列或可迭代对象转换为数据项和顺序均不变的列表。例如，list((1, 2, 3)) 会生成 [1, 2, 3] 而 list('abc') 则会生成 ['a', 'b', 'c']。如果参数即为列表，则会像 seq[:] 那样创建一个副本。

2.5.2 什么是负数索引？

Python 序列的索引可以是正数或负数。索引为正数时，0 是第一个索引值，1 为第二个，依此类推。索引为负数时，-1 为倒数第一个索引值，-2 为倒数第二个，依此类推。可以认为 seq[-n] 就相当于 seq[len(seq)-n]。

使用负数序号有时会很方便。例如 S[:-1] 就是原字符串去掉最后一个字符，这可以用来移除某个字符串末尾的换行符。

2.5.3 序列如何以逆序遍历？

使用内置函数 reversed()：

```
for x in reversed(sequence):
    ...  # do something with x ...
```

原序列不会变化，而是构建一个逆序的新副本以供遍历。

2.5.4 如何从列表中删除重复项？

许多完成此操作的的详细介绍，可参阅 Python Cookbook：

https://code.activestate.com/recipes/52560/

如果列表允许重新排序，不妨先对其排序，然后从列表末尾开始扫描，依次删除重复项：

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

如果列表的所有元素都能用作集合的键（即都是*hashable* ），以下做法速度往往更快：

```
mylist = list(set(mylist))
```

以上操作会将列表转换为集合，从而删除重复项，然后返回成列表。

2.5.5 如何从列表中删除多个项？

类似于删除重复项，一种做法是反向遍历并根据条件删除。不过更简单快速的做法就是切片替换操作，采用隐式或显式的正向迭代遍历。以下是三种变体写法：

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

列表推导式可能是最快的。

2.5.6 如何在 Python 中创建数组？

用列表：

```
["this", 1, "is", "an", "array"]
```

列表在时间复杂度方面相当于 C 或 Pascal 的数组；主要区别在于，Python 列表可以包含多种不同类型的对象。

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that NumPy and other third party packages define array-like structures with various characteristics as well.

若要得到 Lisp 风格的列表，可以用元组模拟 cons 元素：

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

若要具备可变性，可以不用元组而是用列表。模拟 lisp car 函数的是 `lisp_list[0]` ，模拟 cdr 函数的是 `lisp_list[1]` 。仅当真正必要时才会这么用，因为通常这种用法要比 Python 列表慢得多。

2.5.7 如何创建多维列表？

多维数组或许会用以下方式建立：

```
>>> A = [[None] * 2] * 3
```

打印出来貌似没错：

```
>>> A
[[None, None], [None, None], [None, None]]
```

但如果给某一项赋值，结果会同时在多个位置体现出来：

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

原因在于用 * 对列表执行重复操作并不会创建副本，而只是创建现有对象的引用。*3 创建的是包含 3 个引用的列表，每个引用指向的是同一个长度为 2 的列表。1 处改动会体现在所有地方，这一定不是应有的方案。

推荐做法是先创建一个所需长度的列表，然后将每个元素都填充为一个新建列表。

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

以上生成了一个包含 3 个列表的列表，每个子列表的长度为 2。也可以采用列表推导式：

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

或者你还可以使用提供矩阵类型的扩展包；其中最著名的是 NumPy。

2.5.8 如何将方法应用于一系列对象？

可以使用列表推导式：

```
result = [obj.method() for obj in mylist]
```

2.5.9 为什么 a_tuple[i] += ['item'] 会引发异常？

这是由两个因素共同导致的，一是增强赋值运算符属于 赋值运算符，二是 Python 可变和不可变对象之间的差别。

只要元组的元素指向可变对象，这时对元素进行增强赋值，那么这里介绍的内容都是适用的。在此只以 list 和 += 举例。

如果你写成这样:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

触发异常的原因显而易见：1 会与指向（1）的对象 a_tuple[0] 相加，生成结果对象 2，但在试图将运算结果 2 赋值给元组的 0 号元素时就会报错，因为元组元素的指向无法更改。

其实在幕后，上述增强赋值语句的执行过程大致如下：

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

由于元组是不可变的，因此赋值这步会引发错误。

如果写成以下这样：

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

这时触发异常会令人略感惊讶，更让人吃惊的是虽有报错，但加法操作却生效了：

```
>>> a_tuple[0]
['foo', 'item']
```

要明白为何会这样，需要知道 (a) 如果一个对象实现了 `__iadd__` 魔法方法，在执行 += 增强赋值时就会调用它，并采纳其返回值；(b) 对于列表而言，`__iadd__` 相当于在列表上调用 `extend` 并返回该列表。因此对于列表可以说 += 就是 `list.extend` 的“快捷方式”：

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

这相当于：

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

a_list 所引用的对象已被修改，而引用被修改对象的指针又重新被赋值给 `a_list`。赋值的最终结果没有变化，因为它是引用 `a_list` 之前所引用的同一对象的指针，但仍然发生了赋值操作。

因此，在此元组示例中，发生的事情等同于：

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

`__iadd__` 成功执行，因此列表得到了扩充，但是虽然 `result` 指向了 `a_tuple[0]` 已经指向的同一对象，最后的赋值仍然导致了报错，因为元组是不可变的。

2.5.10 我想做一个复杂的排序：能用 Python 进行施瓦茨变换吗？

归功于 Perl 社区的 Randal Schwartz，该技术根据度量值对列表进行排序，该度量值将每个元素映射为“顺序值”。在 Python 中，请利用 `list.sort()` 方法的 `key` 参数：

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.11 如何根据另一个列表的值对某列表进行排序？

将它们合并到元组的迭代器中，对结果列表进行排序，然后选择所需的元素。

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

2.6 对象

2.6.1 什么是类？

类是通过执行 class 语句创建的某种对象的类型。创建实例对象时，用 Class 对象作为模板，实例对象既包含了数据（属性），又包含了数据类型特有的代码（方法）。

类可以基于一个或多个其他类（称之为基类）进行创建。基类的属性和方法都得以继承。这样对象模型就可以通过继承不断地进行细化。比如通用的 `Mailbox` 类提供了邮箱的基本访问方法.，它的子类 `MboxMailbox`、`MaildirMailbox`、`OutlookMailbox` 则能够处理各种特定的邮箱格式。

2.6.2 什么是方法？

方法是属于对象的函数，对于对象 `x`，通常以 `x.name(arguments...)` 的形式调用。方法以函数的形式给出定义，位于类的定义内：

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 什么是 self ？

Self 只是方法的第一个参数的习惯性名称。假定某个类中有个方法定义为 `meth(self, a, b, c)`，则其实例 `x` 应以 `x.meth(a, b, c)` 的形式进行调用；而被调用的方法则应视其为做了 `meth(x, a, b, c)` 形式的调用。

另请参阅*为什么必须在方法定义和调用中显式使用“self”？*。

2.6.4 如何检查对象是否为给定类或其子类的一个实例？

可使用内置函数 `isinstance(obj, cls)`。可以检测对象是否属于多个类中某一个的实例，只要把单个类换成元组即可，比如 `isinstance(obj, (class1, class2, ...))`，还可以检查对象是否属于某个 Python 内置类型，例如 `isinstance(obj, str)` 或 `isinstance(obj, (int, float, complex))`。

请注意 `isinstance()` 还会检测派生自*abstract base class* 的虚继承。因此对于已注册的类，即便没有直接或间接继承自抽象基类，对抽象基类的检测都将返回 `True`。要想检测“真正的继承”，请扫描类的*MRO*:

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # direct
True
>>> isinstance(c, P)        # indirect
True
>>> isinstance(c, Mapping)  # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

请注意，大多数程序不会经常用 `isinstance()` 对用户自定义类进行检测。如果是自己开发的类，更合适的面向对象编程风格应该是在类中定义多种方法，以封装特定的行为，而不是检查对象属于什么类再据此干不同的事。假定有如下执行某些操作的函数:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

更好的方法是在所有类上定义一个 `search()` 方法，然后调用它:

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox

class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

2.6.5 什么是委托？

委托是一种面向对象的技术（也称为设计模式）。假设对象 x 已经存在，现在想要改变其某个方法的行为。可以创建一个新类，其中提供了所需修改方法的新实现，而将所有其他方法都委托给 x 的对应方法。

Python 程序员可以轻松实现委托。比如以下实现了一个类似于文件的类，只是会把所有写入的数据转换为大写:

```
class UpperOut:
```

(下页继续)

(繼續上一頁)

```
    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

这里 UpperOut 类重新定义了 write() 方法，在调用下层的 self._outfile.write() 方法之前，会将参数字符串转换为大写。其他所有方法则都被委托给下层的 self._outfile 对象。委托是通过 __getattr__ 方法完成的；请参阅 语言参考了解有关控制属性访问的更多信息。

请注意，更常见情况下，委托可能会变得比较棘手。如果属性既需要写入又需要读取，那么类还必须定义 __setattr__() 方法，而这时就必须十分的小心。基础的 __setattr__() 实现代码大致如下：

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

大多数 __setattr__() 实现必须修改 self.__dict__ 来为自身保存局部状态，而不至于引起无限递归。

2.6.6 如何在派生类中调用被重载的基类方法?

使用内置的 super() 函数：

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

如果是 Python 3.0 之前的版本，可能用的还是传统类：对于诸如 class Derived(Base): ... 之类的类定义，可以用 Base.meth(self, arguments...) 的形式调用 Base (或 Base 的某个基类) 中定义的方法 meth() 。这里，Base.meth 是一个未绑定的方法，因此需要给出 self 参数。

2.6.7 如何让代码更容易对基类进行修改？

可以为基类赋一个别名并基于该别名进行派生。这样只要修改赋给该别名的值即可。顺便提一下，如要动态地确定（例如根据可用的资源）该使用哪个基类，这个技巧也非常方便。例如：

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

2.6.8 如何创建静态类数据和静态类方法？

Python 支持静态数据和静态方法（以 C++ 或 Java 的定义而言）。

静态数据只需定义一个类属性即可。若要为属性赋新值，则必须在赋值时显式使用类名：

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

对于所有符合 `isinstance(c, C)` 的 `c`，`c.count` 也同样指向 `C.count`，除非被 `c` 自身重载，或者被从 `c.__class__` 回溯到基类 `C` 的搜索路径上的某个类所重载。

注意：在 C 的某个方法内部，像 `self.count = 42` 这样的赋值将在 `self` 自身的字典中新建一个名为"count" 的不相关实例。想要重新绑定类静态数据名称就必须总是指明类名，无论是在方法内部还是外部：

```
C.count = 314
```

Python 支持静态方法：

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

不过为了获得静态方法的效果，还有一种做法直接得多，也即使用模块级函数即可：

```
def getcount():
    return C.count
```

如果代码的结构化比较充分，每个模块只定义了一个类（或者多个类的层次关系密切相关），那就具备了应有的封装。

2.6.9 在 Python 中如何重载构造函数（或方法）？

这个答案实际上适用于所有方法，但问题通常首先出现于构造函数的应用场景中。

在 C++ 中，代码会如下所示：

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

在 Python 中，只能编写一个构造函数，并用默认参数捕获所有情况。例如：

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

这不完全等同，但在实践中足够接近。

也可以试试采用变长参数列表，例如：

```
def __init__(self, *args):
    ...
```

上述做法同样适用于所有方法定义。

2.6.10 在用 __spam 的时候得到一个类似 _SomeClassName__spam 的错误信息。

以双下划线打头的变量名会被“破坏”，以便以一种简单高效的方式定义类私有变量。任何形式为 __spam 的标识符（至少前缀两个下划线，至多后缀一个下划线）文本均会被替换为 _classname__spam，其中 classname 为去除了全部前缀下划线的当前类名称。

这并不能保证私密性：外部用户仍然可以访问”_classname__spam” 属性，私有变量值也在对象的 __dict__ 中可见。许多 Python 程序员根本不操心要去使用私有变量名。

2.6.11 类定义了 __del__ 方法，但是删除对象时没有调用它。

这有几个可能的原因。

del 语句不一定调用 __del__() ——它只是减少对象的引用计数，如果（引用计数）达到零，才会调用 __del__()。

如果数据结构包含循环链接（比如树的每个子节点都带有父节点的引用，而每个父节点也带有子节点的列表），则引用计数永远不会回零。尽管 Python 偶尔会用某种算法检测这种循环引用，但在数据结构的最后一条引用消失之后，垃圾收集器可能还要过段时间才会运行，因此 __del__() 方法可能会在不方便和随机的时刻被调用。这对于重现一个问题，是非常不方便的。更糟糕的是，各个对象的 __del__() 方法是以随机顺序执行的。虽然可以运行 gc.collect() 来强制执行垃圾回收工作，但 仍会存在一些对象永远不会被回收的失控情况。

尽管有垃圾回收器的存在，但为对象定义显式的 close() 方法，只要一用完即可供调用，这依然是一个好主意。这样 close() 方法即可删除引用子对象的属性。请勿直接调用 __del__() ——而 __del__() 应该调用 close()，并且应能确保可以对同一对象多次调用 close() 。

另一种避免循环引用的做法是利用 weakref 模块，该模块允许指向对象但不增加其引用计数。例如，树状数据结构应该对父节点和同级节点使用弱引用（如果真要用的话！）

最后提一下，如果 __del__() 方法引发了异常，会将警告消息打印到 sys.stderr 。

2.6.12 如何获取给定类的所有实例的列表？

Python 不会记录类（或内置类型）的实例。可以在类的构造函数中编写代码，通过保留每个实例的弱引用列表来跟踪所有实例。

2.6.13 为什么 id() 的结果看起来不是唯一的？

id() 返回一个整数，该整数在对象的生命周期内保证是唯一的。因为在 CPython 中，这是对象的内存地址，所以经常发生在从内存中删除对象之后，下一个新创建的对象被分配在内存中的相同位置。这个例子说明了这一点：

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

这两个 id 属于不同的整数对象，之前先创建了对象，执行 id() 调用后又立即被删除了。若要确保检测 id 时的对象仍处于活动状态，请再创建一个对该对象的引用：

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.6.14 什么情况下可以依靠 *is* 运算符进行对象的身份相等性测试？

is 运算符可用于测试对象的身份相等性。a is b 等价于 id(a) == id(b)。

身份相等性最重要的特性就是对象总是等同于自身，a is a 一定返回 True。身份相等性测试的速度通常比相等性测试要快。而且与相等性测试不一样，身份相等性测试会确保返回布尔值 True 或 False。

但是，身份相等性测试 只能在对象身份确定的场景下才可替代相等性测试。一般来说，有以下 3 种情况对象身份是可以确定的：

1）赋值操作创建了新的名称但没有改变对象身份。在赋值操作 new = old 之后，可以保证 new is old。

2）将对象置入存放对象引用的容器，对象身份不会改变。在列表赋值操作 s[0] = x 之后，可以保证 s[0] is x。

3）单例对象，也即该对象只能存在一个实例。在赋值操作 a = None 和 b = None 之后，可以保证 a is b，因为 None 是单例对象。

其他大多数情况下，都不建议使用身份相等性测试，而应采用相等性测试。尤其是不应将身份相等性测试用于检测常量值，例如 int 和 str，因为他们并不一定是单例对象：

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

同样地，可变容器的新实例，对象身份一定不同：

```
>>> a = []
>>> b = []
>>> a is b
False
```

在标准库代码中，给出了一些正确使用对象身份测试的常见模式：

1）正如 **PEP 8** 所推荐的，对象身份测试是 None 值的推荐检测方式。这样的代码读起来就像自然的英文，并可以避免与其他可能为布尔值且计算结果为 False 的对象相混淆。

2）如果 None 也是有效的输入值，则可选参数的检测就可能比较棘手。这时可以创建一个单例哨兵对象，要确保与其他对象都不相同。例如，下面是一个类似 dict.pop() 的方法：

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
```

(下页继续)

(繼續上一頁)

```
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3）编写容器的实现代码时，有时需要用对象身份测试来加强相等性检测。这样代码就不会被 `float('NaN')` 这类与自身不相等的对象所干扰。

例如，以下是 `collections.abc.Sequence.__contains__()` 的实现代码：

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

2.6.15 How can a subclass control what data is stored in an immutable instance?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

All of these immutable classes have a different signature than their parent class:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

The classes can be used like this:

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

2.7 模块

2.7.1 如何创建.pyc 文件？

当首次导入模块时（或当前已编译文件创建之后源文件发生了改动），在 “.py“文件所在目录的 “__pycache__“子目录下会创建一个包含已编译代码的 “.pyc“文件。该 “.pyc“文件的名称开头部分将与 “.py“文件名相同，并以 “.pyc“为后缀，中间部分则依据创建它的 “python“版本而各不相同。（详见 **PEP 3147**。）

.pyc 文件有可能会无法创建，原因之一是源码文件所在的目录存在权限问题，这样就无法创建 __pycache__ 子目录。假如以某个用户开发程序而以另一用户运行程序，就有可能发生权限问题，测试 Web 服务器就属于这种情况。

除非设置了 PYTHONDONTWRITEBYTECODE 环境变量，否则导入模块并且 Python 能够创建 “__pycache__“子目录并把已编译模块写入该子目录（权限、存储空间等等）时，.pyc 文件就将自动创建。

在最高层级运行的 Python 脚本不会被视为经过了导入操作，因此不会创建 .pyc 文件。假定有一个最高层级的模块文件 foo.py，它导入了另一个模块 xyz.py，当运行 foo 模块（通过输入 shell 命令 python foo.py ），则会为 xyz 创建一个 .pyc，因为 xyz 是被导入的，但不会为 foo 创建 .pyc 文件，因为 foo.py 不是被导入的。

若要为 foo 创建 .pyc 文件——即为未做导入的模块创建 .pyc 文件——可以利用 py_compile 和 compileall 模块。

py_compile 模块能够手动编译任意模块。一种做法是交互式地使用该模块中的 compile() 函数:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

这将会将 .pyc 文件写入与 foo.py 相同位置下的 __pycache__ 子目录（或者你也可以通过可选参数 cfile 来重载该行为）。

还可以用 compileall 模块自动编译一个或多个目录下的所有文件。只要在命令行提示符中运行 compileall.py 并给出要编译的 Python 文件所在目录路径即可:

```
python -m compileall .
```

2.7.2 如何找到当前模块名称？

模块可以查看预定义的全局变量 __name__ 获悉自己的名称。如其值为 '__main__' ，程序将作为脚本运行。通常，许多通过导入使用的模块同时也提供命令行接口或自检代码，这些代码只在检测到处于 __name__ 之后才会执行:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 如何让模块相互导入？

假设有以下模块：

foo.py：

```
from bar import bar_var
foo_var = 1
```

bar.py：

```
from foo import foo_var
bar_var = 2
```

问题是解释器将执行以下步骤：

- 首先导入 foo
- 创建用于 foo 的空全局变量
- foo 被编译并开始执行
- foo 导入 bar
- 创建了用于 bar 的空全局变量
- bar 被编译并开始执行
- bar 导入 foo（这是一个空操作，因为已经有一个名为 foo 的模块）。
- bar.foo_var = foo.foo_var

最后一步失败了，因为 Python 还没有完成对 foo 的解释，foo 的全局符号字典仍然是空的。

当你使用 `import foo`，然后尝试在全局代码中访问 `foo.foo_var` 时，会发生同样的事情。

这个问题有（至少）三种可能的解决方法。

Guido van Rossum 建议完全避免使用 `from <module> import ...`，并将所有代码放在函数中。全局变量和类变量的初始化只应使用常量或内置函数。这意味着导入模块中的所有内容都以 `<module>.<name>` 的形式引用。

Jim Roskind 建议每个模块都应遵循以下顺序：

- 导出（全局变量、函数和不需要导入基类的类）
- `import` 语句
- 本模块的功能代码（包括根据导入值进行初始化的全局变量）。

van Rossum 不喜欢以上做法，因为 import 语句出现在了奇怪的地方，但确实有效。

Matthias Urlichs 建议对代码进行重构，使得递归导入根本就没必要发生。

这些解决方案并不相互排斥。

2.7.4 __import__('x.y.z') 返回的是 <module 'x'>；该如何得到 z 呢？

不妨考虑换用 `importlib` 中的函数 `import_module()`：

```
z = importlib.import_module('x.y.z')
```

2.7.5 对已导入的模块进行了编辑并重新导入，但变动没有得以体现。这是为什么？

出于效率和一致性的原因，Python 仅在第一次导入模块时读取模块文件。否则，在一个多模块的程序中，每个模块都会导入相同的基础模块，那么基础模块将会被一而再、再而三地解析。如果要强行重新读取已更改的模块，请执行以下操作：

```
import importlib
import modname
importlib.reload(modname)
```

警告：这种技术并非万无一失。尤其是模块包含了以下语句时：

```
from modname import some_objects
```

仍将继续使用前一版的导入对象。如果模块包含了类的定义，并 不会用新的类定义更新现有的类实例。这样可能会导致以下矛盾的行为：

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

只要把类对象的 id 打印出来，问题的性质就会一目了然：

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

CHAPTER 3

设计和历史常见问题

3.1 为什么 Python 使用缩进来分组语句？

Guido van Rossum 认为使用缩进进行分组非常优雅，并且大大提高了普通 Python 程序的清晰度。大多数人在一段时间后就学会并喜欢上这个功能。

由于没有开始/结束括号，因此解析器感知的分组与人类读者之间不会存在分歧。偶尔 C 程序员会遇到像这样的代码片段:

```
if (x <= y)
        x++;
        y--;
z++;
```

如果条件为真，则只执行 x++ 语句，但缩进会使你认为情况并非如此。即使是经验丰富的 C 程序员有时也会长久地盯着它发呆，不明白为什么在 x > y`` 时 ``y 也会减少。

因为没有开始/结束花括号，所以 Python 更不容易发生编码风格冲突。在 C 中有许多不同的放置花括号的方式。在习惯了阅读和编写某种特定风格的代码之后，当阅读（或被要求编写）另一种风格的代码时通常都会令人感觉有点不舒服）。

许多编码风格将开始/结束括号单独放在一行上。这使得程序相当长，浪费了宝贵的屏幕空间，使得更难以对程序进行全面的了解。理想情况下，函数应该适合一个屏幕（例如，20--30 行）。20 行 Python 可以完成比 20 行 C 更多的工作。这不仅仅是由于缺少开始/结束括号 -- 缺少声明和高级数据类型也是其中的原因 -- 但缩进基于语法肯定有帮助。

3.2 为什么简单的算术运算得到奇怪的结果？

請見下一個問題。

3.3 F何浮點數運算如此不精確?

用户经常对这样的结果感到惊讶:

```
>>> 1.2 - 1.0
0.19999999999999996
```

并且认为这是 Python 中的一个 bug。其实不是这样。这与 Python 关系不大，而与底层平台如何处理浮点数字关系更大。

CPython 中的 `float` 类型使用 C 语言的 `double` 类型进行存储。`float` 对象的值是以固定的精度（通常为 53 位）存储的二进制浮点数，由于 Python 使用 C 操作，而后者依赖于处理器中的硬件实现来执行浮点运算。这意味着就浮点运算而言，Python 的行为类似于许多流行的语言，包括 C 和 Java。

许多可以轻松地用十进制表示的数字不能用二进制浮点表示。例如，在输入以下语句后:

```
>>> x = 1.2
```

为 `x` 存储的值是与十进制的值 `1.2` (非常接近) 的近似值，但不完全等于它。在典型的机器上，实际存储的值是:

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

它对应于十进制数值:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

典型的 53 位精度为 Python 浮点数提供了 15-16 位小数的精度。

要获得更完整的解释，请参阅 Python 教程中的 浮点算术一章。

3.4 F何 Python 字串F不可變動

有許多優點。

一个是性能：知道字符串是不可变的，意味着我们可以在创建时为它分配空间，并且存储需求是固定不变的。这也是元组和列表之间区别的原因之一。

另一个优点是，Python 中的字符串被视为与数字一样 “基本”。任何动作都不会将值 8 更改为其他值，在 Python 中，任何动作都不会将字符串”8” 更改为其他值。

3.5 为什么必须在方法定义和调用中显式使用 “self”?

此構想從 Modula-3 而來。有許多原因可以F是非常實用。

首先，更明显的显示出，使用的是方法或实例属性而不是局部变量。阅读 `self.x` 或 `self.meth()` 可以清楚地表明，即使您不知道类的定义，也会使用实例变量或方法。在 C++ 中，可以通过缺少局部变量声明来判断（假设全局变量很少见或容易识别）——但是在 Python 中没有局部变量声明，所以必须查找类定义才能确定。一些 C++ 和 Java 编码标准要求实例属性具有 `m_` 前缀，因此这种显式性在这些语言中仍然有用。

其次，这意味着如果要显式引用或从特定类调用该方法，不需要特殊语法。在 C++ 中，如果你想使用在派生类中重写基类中的方法，你必须使用 `::` 运算符 -- 在 Python 中你可以编写 `baseclass.methodname(self, <argument list>)`。这对于 `__init__()` 方法非常有用，特别是在派生类方法想要扩展同名的基类方法，而必须以某种方式调用基类方法时。

最后，它解决了变量赋值的语法问题：为了 Python 中的局部变量（根据定义!）在函数体中赋值的那些变量（并且没有明确声明为全局）赋值，就必须以某种方式告诉解释器一个赋值是为了分配一个实例变量而不是一个局部变量，它最好是通过语法实现的（出于效率原因）。C++ 通过声明来做到这一点，但是

Python 没有声明，仅仅为了这个目的而引入它们会很可惜。使用显式的 `self.var` 很好地解决了这个问题。类似地，对于使用实例变量，必须编写 `self.var` 意味着对方法内部的非限定名称的引用不必搜索实例的目录。换句话说，局部变量和实例变量存在于两个不同的命名空间中，您需要告诉 Python 使用哪个命名空间。

3.6 为什么不能在表达式中赋值？

自 Python 3.8 开始，你能做到的！

赋值表达式使用海象运算符 := 在表达式中为变量赋值:

```
while chunk := fp.read(200):
    print(chunk)
```

请参阅 **PEP 572** 了解详情。

3.7 为什么 Python 对某些功能（例如 list.index()）使用方法来实现，而其他功能（例如 len(List)）使用函数实现？

正如 Guido 所说：

> (a) 对于某些操作，前缀表示法比后缀更容易阅读 -- 前缀（和中缀！）运算在数学中有着悠久的传统，就像在视觉上帮助数学家思考问题的记法。比较一下我们将 x*(a+b) 这样的公式改写为 x*a+x*b 的容易程度，以及使用原始 OO 符号做相同事情的笨拙程度。
>
> (b) 当读到写有 len(X) 的代码时，就知道它要求的是某件东西的长度。这告诉我们两件事：结果是一个整数，参数是某种容器。相反，当阅读 x.len() 时，必须已经知道 x 是某种实现接口的容器，或者是从具有标准 len() 的类继承的容器。当没有实现映射的类有 get() 或 key() 方法，或者不是文件的类有 write() 方法时，我们偶尔会感到困惑。
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 为什么 join() 是一个字符串方法而不是列表或元组方法？

从 Python 1.6 开始，字符串变得更像其他标准类型，当添加方法时，这些方法提供的功能与始终使用 String 模块的函数时提供的功能相同。这些新方法中的大多数已被广泛接受，但似乎让一些程序员感到不舒服的一种方法是：

```
", ".join(['1', '2', '4', '8', '16'])
```

结果如下:

```
"1, 2, 4, 8, 16"
```

反对这种用法有两个常见的论点。

第一条是这样的：“使用字符串文本 (String Constant) 的方法看起来真的很难看”，答案是也许吧，但是字符串文本只是一个固定值。如果在绑定到字符串的名称上允许使用这些方法，则没有逻辑上的理由使其在文字上不可用。

第二个异议通常是这样的：“我实际上是在告诉序列使用字符串常量将其成员连接在一起”。遗憾的是并非如此。出于某种原因，把 `split()` 作为一个字符串方法似乎要容易得多，因为在这种情况下，很容易看到:

```
"1, 2, 4, 8, 16".split(", ")
```

是对字符串文本的指令，用于返回由给定分隔符分隔的子字符串（或在默认情况下，返回任意空格）。

`join()` 是字符串方法，因为在使用该方法时，您告诉分隔符字符串去迭代一个字符串序列，并在相邻元素之间插入自身。此方法的参数可以是任何遵循序列规则的对象，包括您自己定义的任何新的类。对于字节和字节数组对象也有类似的方法。

3.9 异常有多快？

如果没有引发异常，则 try/except 块的效率极高。实际上捕获异常是昂贵的。在 2.0 之前的 Python 版本中，通常使用这个习惯用法:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

只有当你期望 dict 在任何时候都有 key 时，这才有意义。如果不是这样的话，你就是应该这样编码:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

对于这种特定的情况，您还可以使用 `value = dict.setdefault(key, getvalue(key))`，但前提是调用 `getvalue()` 足够便宜，因为在所有情况下都会对其进行评估。

3.10 为什么 Python 中没有 switch 或 case 语句？

你可以通过一系列 `if... elif... elif... else`. 轻松完成这项工作。对于 switch 语句语法已经有了一些建议，但尚未就是否以及如何进行范围测试达成共识。有关完整的详细信息和当前状态，请参阅 **PEP 275** 。

对于需要从大量可能性中进行选择的情况，可以创建一个字典，将 case 值映射到要调用的函数。例如:

```
def function_1(...):
    ...

functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1, ...}

func = functions[value]
func()
```

对于对象调用方法，可以通过使用 `getattr()` 内置检索具有特定名称的方法来进一步简化:

```
def visit_a(self, ...):
    ...
...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

建议对方法名使用前缀，例如本例中的 `visit_` 。如果没有这样的前缀，如果值来自不受信任的源，攻击者将能够调用对象上的任何方法。

3.11 难道不能在解释器中模拟线程，而非得依赖特定于操作系统的线程实现吗？

答案 1：不幸的是，解释器为每个 Python 堆栈帧推送至少一个 C 堆栈帧。此外，扩展可以随时回调 Python。因此，一个完整的线程实现需要对 C 的线程支持。

答案 2：幸运的是，Stackless Python 有一个完全重新设计的解释器循环，可以避免 C 堆栈。

3.12 🄵何 lambda 表示式不能包含在🄵述🄵

Python 的 lambda 表达式不能包含语句，因为 Python 的语法框架不能处理嵌套在表达式内部的语句。然而，在 Python 中，这并不是一个严重的问题。与其他语言中添加功能的 lambda 表单不同，Python 的 lambdas 只是一种速记符号，如果您懒得定义函数的话。

函数已经是 Python 中的第一类对象，可以在本地范围内声明。因此，使用 lambda 而不是本地定义的函数的唯一优点是你不需要为函数创建一个名称 -- 这只是一个分配了函数对象 (与 lambda 表达式生成的对象类型完全相同) 的局部变量！

3.13 Python 可以被編譯成機器語言或 C 語言或其他種語言嗎?

Cython 将带有可选注释的 Python 修改版本编译到 C 扩展中。Nuitka 是一个将 Python 编译成 C++ 代码的新兴编译器，旨在支持完整的 Python 语言。要编译成 Java，可以考虑 VOC 。

3.14 Python 如何管理記憶體?

Python 内存管理的细节取决于实现。Python 的标准实现*CPython* 使用引用计数来检测不可访问的对象，并使用另一种机制来收集引用循环，定期执行循环检测算法来查找不可访问的循环并删除所涉及的对象。`gc` 模块提供了执行垃圾回收、获取调试统计信息和优化收集器参数的函数。

但是，其他实现 (如 Jython 或 PyPy)，) 可以依赖不同的机制，如完全的垃圾回收器 。如果你的 Python 代码依赖于引用计数实现的行为，则这种差异可能会导致一些微妙的移植问题。

在一些 Python 实现中，以下代码（在 CPython 中工作的很好）可能会耗尽文件描述符:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

实际上，使用 CPython 的引用计数和析构函数方案，每个新赋值的 *f* 都会关闭前一个文件。然而，对于传统的 GC，这些文件对象只能以不同的时间间隔（可能很长的时间间隔）被收集（和关闭）。

如果要编写可用于任何 python 实现的代码，则应显式关闭该文件或使用 `with` 语句；无论内存管理方案如何，这都有效:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 F何 CPython 不使用更多傳統的垃圾回收機制?

首先，这不是 C 标准特性，因此不能移植。(是的，我们知道 Boehm GC 库。它包含了 大多数常见平台（但不是所有平台）的汇编代码，尽管它基本上是透明的，但也不是完全透明的; 要让 Python 使用它，需要使用补丁。)

当 Python 嵌入到其他应用程序中时，传统的 GC 也成为一个问题。在独立的 Python 中，可以用 GC 库提供的版本替换标准的 malloc() 和 free()，嵌入 Python 的应用程序可能希望用 它自己替代 malloc() 和 free()，而可能不需要 Python 的。现在，CPython 可以正确地实现 malloc() 和 free()。

3.16 當 CPython 結束時，F何所有的記憶體不會被釋放?

当 Python 退出时，从全局命名空间或 Python 模块引用的对象并不总是被释放。如果存在循环引用，则可能发生这种情况 C 库分配的某些内存也是不可能释放的（例如像 Purify 这样的工具会抱怨这些内容）。但是，Python 在退出时清理内存并尝试销毁每个对象。

如果要强制 Python 在释放时删除某些内容，请使用 `atexit` 模块运行一个函数，强制删除这些内容。

3.17 为什么有单独的元组和列表数据类型？

虽然列表和元组在许多方面是相似的，但它们的使用方式通常是完全不同的。可以认为元组类似于 Pascal 记录或 C 结构；它们是相关数据的小集合，可以是不同类型的数据，可以作为一个组进行操作。例如，笛卡尔坐标适当地表示为两个或三个数字的元组。

另一方面，列表更像其他语言中的数组。它们倾向于持有不同数量的对象，所有对象都具有相同的类型，并且逐个操作。例如，`os.listdir('.')` 返回表示当前目录中的文件的字符串列表。如果向目录中添加了一两个文件，对此输出进行操作的函数通常不会中断。

元组是不可变的，这意味着一旦创建了元组，就不能用新值替换它的任何元素。列表是可变的，这意味着您始终可以更改列表的元素。只有不变元素可以用作字典的 key，因此只能将元组和非列表用作 key。

3.18 列表是如何在 CPython 中实现的？

CPython 的列表实际上是可变长度的数组，而不是 lisp 风格的链表。该实现使用对其他对象的引用的连续数组，并在列表头结构中保留指向该数组和数组长度的指针。

这使得索引列表 `a[i]` 的操作成本与列表的大小或索引的值无关。

当添加或插入项时，将调整引用数组的大小。并采用了一些巧妙的方法来提高重复添加项的性能; 当数组必须增长时，会分配一些额外的空间，以便在接下来的几次中不需要实际调整大小。

3.19 字典是如何在 CPython 中实现的？

CPython 的字典实现为可调整大小的哈希表。与 B-树相比，这在大多数情况下为查找（目前最常见的操作）提供了更好的性能，并且实现更简单。

字典的工作方式是使用 `hash()` 内置函数计算字典中存储的每个键的 hash 代码。hash 代码根据键和每个进程的种子而变化很大；例如，"Python" 的 hash 值为-539294296，而"python"(一个按位不同的字符串) 的 hash 值为 1142331976。然后，hash 代码用于计算内部数组中将存储该值的位置。假设您存储的键都具有不同的 hash 值，这意味着字典需要恒定的时间 -- O(1)，用 Big-O 表示法 -- 来检索一个键。

3.20 为什么字典 key 必须是不可变的？

字典的哈希表实现使用从键值计算的哈希值来查找键。如果键是可变对象，则其值可能会发生变化，因此其哈希值也会发生变化。但是，由于无论谁更改键对象都无法判断它是否被用作字典键值，因此无法在字典中修改条目。然后，当你尝试在字典中查找相同的对象时，将无法找到它，因为其哈希值不同。如果你尝试查找旧值，也不会找到它，因为在该哈希表中找到的对象的值会有所不同。

如果你想要一个用列表索引的字典，只需先将列表转换为元组；用函数 `tuple(L)` 创建一个元组，其条目与列表 `L` 相同。元组是不可变的，因此可以用作字典键。

已经提出的一些不可接受的解决方案：

- 哈希按其地址（对象 ID）列出。这不起作用，因为如果你构造一个具有相同值的新列表，它将无法找到；例如:

  ```
  mydict = {[1, 2]: '12'}
  print(mydict[[1, 2]])
  ```

  会引发一个 `KeyError` 异常，因为第二行中使用的 `[1, 2]` 的 id 与第一行中的 id 不同。换句话说，应该使用 == 来比较字典键，而不是使用 `is` 。

- 使用列表作为键时进行复制。这没有用的，因为作为可变对象的列表可以包含对自身的引用，然后复制代码将进入无限循环。

- 允许列表作为键，但告诉用户不要修改它们。当你意外忘记或修改列表时，这将产生程序中的一类难以跟踪的错误。它还使一个重要的字典不变量无效：`d.keys()` 中的每个值都可用作字典的键。

- 将列表用作字典键后，应标记为其只读。问题是，它不仅仅是可以改变其值的顶级对象；你可以使用包含列表作为键的元组。将任何内容作为键关联到字典中都需要将从那里可到达的所有对象标记为只读——并且自引用对象可能会导致无限循环。

如果需要，可以使用以下方法来解决这个问题，但使用它需要你自担风险：你可以将一个可变结构包装在一个类实例中，该实例同时具有 `__eq__()` 和 `__hash__()` 方法。然后，你必须确保驻留在字典（或其他基于 hash 的结构）中的所有此类包装器对象的哈希值在对象位于字典（或其他结构）中时保持固定。:

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

注意，哈希计算由于列表的某些成员可能不可用以及算术溢出的可能性而变得复杂。

此外，必须始终如此，如果 `o1 == o2` （即 `o1.__eq__(o2) is True` ）则 `hash(o1) == hash(o2)`` （即 ``o1.__hash__() == o2.__hash__()` ），无论对象是否在字典中。如果你不能满足这些限制，字典和其他基于 hash 的结构将会出错。

对于 ListWrapper ，只要包装器对象在字典中，包装列表就不能更改以避免异常。除非你准备好认真考虑需求以及不正确地满足这些需求的后果，否则不要这样做。请留意。

3.21 🄵何 list.sort() 不是回傳排序過的串列?

在性能很重要的情况下，仅仅为了排序而复制一份列表将是一种浪费。因此，`list.sort()` 对列表进行了适当的排序。为了提醒您这一事实，它不会返回已排序的列表。这样，当您需要排序的副本，但也需要保留未排序的版本时，就不会意外地覆盖列表。

如果要返回新列表，请使用内置 `sorted()` 函数。此函数从提供的可迭代列表中创建新列表，对其进行排序并返回。例如，下面是如何迭代遍历字典并按 keys 排序:

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

3.22 如何在 Python 中指定和实施接口规范？

由 C++ 和 Java 等语言提供的模块接口规范描述了模块的方法和函数的原型。许多人认为接口规范的编译时强制执行有助于构建大型程序。

Python 2.6 添加了一个 `abc` 模块，允许定义抽象基类 (ABCs)。然后可以使用 `isinstance()` 和 `issubclass()` 来检查实例或类是否实现了特定的 ABC。`collections.abc` 模块定义了一组有用的 ABCs 例如 `Iterable`，`Container`，和 `MutableMapping`

对于 Python，接口规范的许多好处可以通过组件的适当测试规程来获得。

一个好的模块测试套件既可以提供回归测试，也可以作为模块接口规范和一组示例。许多 Python 模块可以作为脚本运行，以提供简单的“自我测试”。即使是使用复杂外部接口的模块，也常常可以使用外部接口的简单“桩代码（stub）”模拟进行隔离测试。可以使用 `doctest` 和 `unittest` 模块或第三方测试框架来构造详尽的测试套件，以运行模块中的每一行代码。

适当的测试规程可以帮助在 Python 中构建大型的、复杂的应用程序以及接口规范。事实上，它可能会更好，因为接口规范不能测试程序的某些属性。例如，`append()` 方法将向一些内部列表的末尾添加新元素；接口规范不能测试您的 `append()` 实现是否能够正确执行此操作，但是在测试套件中检查这个属性是很简单的。

编写测试套件非常有用，并且你可能希望将你的代码设计为易于测试。一种日益流行的技术是面向测试的开发，它要求在编写任何实际代码之前首先编写测试套件的各个部分。当然 Python 也允许你采用更粗率的方式，不必编写任何测试用例。

3.23 🄵何🄵有 goto 語法?

在 1970 年代人们了解到不受限制的 goto 可能导致混乱得像“意大利面”那样难以理解和修改的代码。在高级语言中，它也是不必要的，只需有实现分支 (在 Python 中是使用 `if` 语句以及 `or`, `and` 和 `if-else` 表达式) 和循环 (使用 `while` 和 `for` 语句，并可能包含 `continue` 和 `break`) 的手段就足够了。

人们还可以使用异常捕获来提供甚至能跨函数调用的“结构化 goto”。许多人认为异常可以方便地模拟 C, Fortran 和其他语言中所有合理使用的“go”或“goto”构造。例如:

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

但是不允许你跳到循环的中间，这通常被认为是滥用 goto。谨慎使用。

3.24 为什么原始字符串（r-strings）不能以反斜杠结尾？

更准确地说，它们不能以奇数个反斜杠结束：结尾处的不成对反斜杠会转义结束引号字符，留下未结束的字符串。

原始字符串的设计是为了方便想要执行自己的反斜杠转义处理的处理器 (主要是正则表达式引擎) 创建输入。此类处理器将不匹配的尾随反斜杠视为错误，因此原始字符串不允许这样做。反过来，允许通过使用引号字符转义反斜杠转义字符串。当 r-string 用于它们的预期目的时，这些规则工作的很好。

如果您正在尝试构建 Windows 路径名，请注意所有 Windows 系统调用都使用正斜杠:

```
f = open("/mydir/file.txt")  # works fine!
```

如果您正在尝试为 DOS 命令构建路径名，请尝试以下示例

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.25 为什么 Python 没有属性赋值的“with”语句？

Python 具有'with' 语句，它能将一个代码块的执行包装起来，在进入和退出代码块时调用特定的代码。有些语言具有这样的结构:

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

在 Python 中，这样的结构是不明确的。

其他语言，如 ObjectPascal、Delphi 和 C++ 使用静态类型，因此可以毫不含糊地知道分配给什么成员。这是静态类型的要点 -- 编译器 总是在编译时知道每个变量的作用域。

Python 使用动态类型。事先不可能知道在运行时引用哪个属性。可以动态地在对象中添加或删除成员属性。这使得无法通过简单的阅读就知道引用的是什么属性：局部属性、全局属性还是成员属性?

例如，采用以下不完整的代码段:

```
def foo(a):
    with a:
        print(x)
```

该代码段假设"a" 必须有一个名为"x" 的成员属性。然而，Python 中并没有告诉解释器这一点。假设"a" 是整数，会发生什么？如果有一个名为"x" 的全局变量，它是否会在 with 块中使用？如您所见，Python 的动态特性使得这样的选择更加困难。

然而，Python 可以通过赋值轻松实现"with" 和类似语言特性（减少代码量）的主要好处。代替:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

写成这样:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

这也具有提高执行速度的副作用，因为 Python 在运行时解析名称绑定，而第二个版本只需要执行一次解析。

3.26 生成器为什么不支持 with 语句？

For technical reasons, a generator used directly as a context manager would not work correctly. When, as is most common, a generator is used as an iterator run to completion, no closing is needed. When it is, wrap it as "contextlib.closing(generator)" in the 'with' statement.

3.27 为什么 if/while/def/class 语句需要冒号？

冒号主要用于增强可读性 (ABC 语言实验的结果之一)。考虑一下这个:

```
if a == b
    print(a)
```

与

```
if a == b:
    print(a)
```

注意第二种方法稍微容易一些。请进一步注意，在这个 FAQ 解答的示例中，冒号是如何设置的；这是英语中的标准用法。

另一个次要原因是冒号使带有语法突出显示的编辑器更容易工作；他们可以寻找冒号来决定何时需要增加缩进，而不必对程序文本进行更精细的解析。

3.28 为什么 Python 在列表和元组的末尾允许使用逗号？

Python 允许您在列表，元组和字典的末尾添加一个尾随逗号:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

這有許多原因可被允許。

如果列表，元组或字典的字面值分布在多行中，则更容易添加更多元素，因为不必记住在上一行中添加逗号。这些行也可以重新排序，而不会产生语法错误。

不小心省略逗号会导致难以诊断的错误。例如:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

这个列表看起来有四个元素，但实际上包含三个: "fee", "fiefoo" 和"fum" 。总是加上逗号可以避免这个错误的来源。

允许尾随逗号也可以使编程代码更容易生成。

CHAPTER 4

函式庫和擴充功能的常見問題

4.1 常見函式問題

4.1.1 如何找到可以用来做 XXX 的模块或应用？

在 代码库参考中查找是否有适合的标准库模块。（如果你已经了解标准库的内容，可以跳过这一步）

对于第三方软件包，请搜索 Python Package Index 或是 Google 等其他搜索引擎。用 “Python” 加上一两个你需要的关键字通常会找到有用的东西。

4.1.2 哪🄵可以找到 math.py (socket.py, regex.py, 等...) 原始檔案

如果找不到模块的源文件，可能它是一个内建的模块，或是使用 C，C++ 或其他编译型语言实现的动态加载模块。这种情况下可能是没有源码文件的，类似 `mathmodule.c` 这样的文件会存放在 C 代码目录中（但不在 Python 目录中）。

有 (至少) 三種 Python 模組：

1) 在 Python 的模組被寫成 (.py)
2) 使用 C 编写的动态加载模块（.dll，.pyd，.so，.sl 等）；
3) 使用 C 编写并链接到解释器的模块，要获取此列表，输入：

```
import sys
print(sys.builtin_module_names)
```

4.1.3 我如何使 Python script 執行在 Unix？

你需要作兩件事: 🄵本程式必須可以被執行而且第一行必須"#!" 開頭後面接上 Python 直譯器的路徑

第一点可以用执行 `chmod +x scriptfile` 或是 `chmod 755 scriptfile` 做到。

第二点有很多种做法，最直接的方式是:

```
#!/usr/local/bin/python
```

在文件第一行，使用你所在平台上的 Python 解释器的路径。

如果你希望脚本不依赖 Python 解释器的具体路径，你也可以使用 **env** 程序。假设你的 Python 解释器所在目录已经添加到了 `PATH` 环境变量中，几乎所有的类 Unix 系统都支持下面的写法:

```
#!/usr/bin/env python
```

不要在 CGI 脚本中这样做。CGI 脚本的 `PATH` 环境变量通常会非常精简，所以你必须使用解释器的完整绝对路径。

有时候，用户的环境变量如果太长，可能会导致 **/usr/bin/env** 执行失败；又或者甚至根本就不存在 env 程序。在这种情况下，你可以尝试使用下面的 hack 方法（来自 Alex Rezinsky）:

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

这样做有一个小小的缺点，它会定义脚本的 __doc__ 字符串。不过可以这样修复:

```
__doc__ = """...Whatever..."""
```

4.1.4 Python 中有 curses/termcap 包吗？

对于类 Unix 系统: 标准 Python 源码发行版会在 Modules 子目录中附带 curses 模块，但默认并不会编译。(注意: 在 Windows 平台下不可用——Windows 中没有 curses 模块。)

`curses` 模块支持基本的 curses 特性，同时也支持 ncurses 和 SYSV curses 中的很多额外功能，比如颜色、不同的字符集支持、填充和鼠标支持。这意味着这个模块不兼容只有 BSD curses 模块的操作系统，但是目前仍在维护的系统应该都不会存在这种情况。

对于 Windows 平台: 使用 consolelib 模块.

4.1.5 Python 中存在类似 C 的 onexit() 函数的东西吗？

`atexit` 模块提供了一个与 C 的 `onexit()` 函数类似的注册函数。

4.1.6 为什么我的信号处理函数不能工作？

最常见的问题是信号处理函数没有正确定义参数列表。它会被这样调用:

```
handler(signum, frame)
```

因此它应当声明为带有两个形参:

```
def handler(signum, frame):
    ...
```

4.2 一般性的工作

4.2.1 我如何測試 Python 程式

Python 带有两个测试框架。doctest 模块从模块的 docstring 中寻找示例并执行，对比输出是否与 docstring 中给出的是否一致。

unittest 模块是一个模仿 Java 和 Smalltalk 测试框架的更棒的测试框架。

为了使测试更容易，你应该在程序中使用良好的模块化设计。程序中的绝大多数功能都应该用函数或类方法封装——有时这样做会有额外惊喜，程序会运行得更快（因为局部变量比全局变量访问要快）。除此之外，程序应该避免依赖可变的局部变量，这会使得测试困难许多。

程序的“全局主逻辑”应该尽量简单：

```
if __name__ == "__main__":
    main_logic()
```

在你的程式主模組的底端

一旦你的程序已经组织为一个函数和类行为的有完整集合，你就应该编写测试函数来检测这些行为。可以将自动执行一系列测试的测试集关联到每个模块。这听起来似乎需要大量的工作，但是由于 Python 是如此简洁灵活因此它会极其容易。你可以通过与“生产代码”同步编写测试函数使编程更为愉快和有趣，因为这将更容易并更早发现代码问题甚至设计缺陷。

程序主模块之外的其他“辅助模块”中可以增加自测试的入口。

```
if __name__ == "__main__":
    self_test()
```

通过使用 Python 实现的“假”接口，即使是需要与复杂的外部接口交互的程序也可以在外部接口不可用时进行测试。

4.2.2 怎样用 docstring 创建文档？

pydoc 模块可以用 Python 源码中的 docstring 创建 HTML 文件。也可以使用 epydoc 来只通过 docstring 创建 API 文档。Sphinx 也可以引入 docstring 的内容。

4.2.3 怎样一次只获取一个按键？

在类 Unix 系统中有多种方案。最直接的方法是使用 curses，但是 curses 模块太大了，难以学习。

4.3 线程相关

4.3.1 程序中怎样使用线程？

一定要使用 threading 模块，不要使用 _thread 模块。threading 模块对 _thread 模块提供的底层线程原语做了更易用的抽象。

Aahz 的非常实用的 threading 教程中有一些幻灯片；可以参阅 http://www.pythoncraft.com/OSCON2001/ 。

4.3.2 我的线程都没有运行，为什么？

一旦主线程退出，所有的子线程都会被杀掉。你的主线程运行得太快了，子线程还没来得及工作。

简单的解决方法是在程序中加一个时间足够长的 sleep，让子线程能够完成运行。

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <---------------------------!
```

但目前（在许多平台上）线程不是并行运行的，而是按顺序依次执行！原因是系统线程调度器在前一个线程阻塞之前不会启动新线程。

简单的解决方法是在运行函数的开始处加一个时间很短的 sleep。

```
def thread_task(name, n):
    time.sleep(0.001)  # <--------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

比起用 `time.sleep()` 猜一个合适的等待时间，使用信号量机制会更好些。有一个办法是使用 `queue` 模块创建一个 queue 对象，让每一个线程在运行结束时 append 一个令牌到 queue 对象中，主线程中从 queue 对象中读取与线程数量一致的令牌数量即可。

4.3.3 如何将任务分配给多个工作线程？

最简单的方式是使用 `concurrent.futures` 模块，特别是其中的 `ThreadPoolExecutor` 类。

或者，如果你想更好地控制分发算法，你也可以自己写逻辑实现。使用 `queue` 模块来创建任务列表队列。`Queue` 类维护一个了一个存有对象的列表，提供了 `.put(obj)` 方法添加元素，并且可以用 `.get()` 方法获取元素。这个类会使用必要的加锁操作，以此确保每个任务只会执行一次。

这是一个简单的例子：

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.currentThread(), end=' ')
            print('queue empty')
```

(下页继续)

(繼續上一頁)

```
            break
        else:
            print('Worker', threading.currentThread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

运行时会产生如下输出：

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

查看模块的文档以获取更多信息；`Queue` 类提供了多种接口。

4.3.4 怎样修改全局变量是线程安全的？

Python VM 内部会使用*global interpreter lock*（GIL）来确保同一时间只有一个线程运行。通常 Python 只会在字节码指令之间切换线程；切换的频率可以通过设置 `sys.setswitchinterval()` 指定。从 Python 程序的角度来看，每一条字节码指令以及每一条指令对应的 C 代码实现都是原子的。

理论上说，具体的结果要看具体的 PVM 字节码实现对指令的解释。而实际上，对内建类型（int，list，dict 等）的共享变量的“类原子”操作都是原子的。

举例来说，下面的操作是原子的（L、L1、L2 是列表，D、D1、D2 是字典，x、y 是对象，i，j 是 int 变量）：

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
```

(下页继续)

(繼續上一頁)

```
D1.update(D2)
D.keys()
```

这些不是原子的：

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

覆盖其他对象的操作会在其他对象的引用计数变成 0 时触发其 `__del__()` 方法，这可能会产生一些影响。对字典和列表进行大量操作时尤其如此。如果有疑问的话，使用互斥锁！

4.3.5 不能删除全局解释器锁吗？

global interpreter lock （GIL）通常被视为 Python 在高端多核服务器上开发时的阻力，因为（几乎）所有 Python 代码只有在获取到 GIL 时才能运行，所以多线程的 Python 程序只能有效地使用一个 CPU。

在 Python 1.5 时代，Greg Stein 开发了一个完整的补丁包（“free threadings” 补丁），移除了 GIL，并用粒度更合适的锁来代替。Adam Olsen 最近也在他的 python-safethread 项目里做了类似的实验。不幸的是，由于为了移除 GIL 而使用了大量细粒度的锁，这两个实验在单线程测试中的性能都有明显的下降（至少慢 30%）。

但这并意味着你不能在多核机器上很好地使用 Python！你只需将任务划分为多 * 进程 *，而不是多 * 线程 *。新的 `concurrent.futures` 模块中的 `ProcessPoolExecutor` 类提供了一个简单的方法；如果你想对任务分发做更多控制，可以使用 `multiprocessing` 模块提供的底层 API。

恰当地使用 C 拓展也很有用；使用 C 拓展处理耗时较久的任务时，拓展可以在线程执行 C 代码时释放 GIL，让其他线程执行。`zlib` 和 `hashlib` 等标准库模块已经这样做了。

也有建议说 GIL 应该是解释器状态锁，而不是完全的全局锁；解释器不应该共享对象。不幸的是，这也不可能发生。由于目前许多对象的实现都有全局的状态，因此这是一个艰巨的工作。举例来说，小整型数和短字符串会缓存起来，这些缓存将不得不移动到解释器状态中。其他对象类型都有自己的自由变量列表，这些自由变量列表也必须移动到解释器状态中。等等。

我甚至怀疑这些工作是否可能在有限的时间内完成，因为同样的问题在第三方拓展中也会存在。第三方拓展编写的速度可比你将它们转换为把全局状态存入解释器状态中的速度快得多。

最后，假设多个解释器不共享任何状态，那么这样做比每个进程一个解释器好在哪里呢?

4.4 输入输出

4.4.1 怎样删除文件？(以及其他文件相关的问题……)

使用 `os.remove(filename)` 或 `os.unlink(filename)`。查看 `os` 模块以获取更多文档。这两个函数是一样的，`unlink()` 是这个函数在 Unix 系统调用中的名字。

如果要删除目录，应该使用 `os.rmdir()`；使用 `os.mkdir()` 创建目录。`os.makedirs(path)` 会创建 `path` 中任何不存在的目录。`os.removedirs(path)` 则会删除其中的目录，只要它们都是空的；如果你想删除整个目录以及其中的内容，可以使用 `shutil.rmtree()`。

重命名文件可以使用 `os.rename(old_path, new_path)`。

如果需要截断文件，使用 `f = open(filename, "rb+")` 打开文件，然后使用 `f.truncate(offset)`；offset 默认是当前的搜索位置。也可以对使用 `os.open()` 打开的文件使用 `os.ftruncate(fd, offset)`，其中 *fd* 是文件描述符（一个小的整型数）。

`shutil` 模块也包含了一些处理文件的函数，包括 `copyfile()`，`copytree()` 和 `rmtree()`。

4.4.2 怎样复制文件？

`shutil` 模块有一个 `copyfile()` 函数。注意在 MacOS 9 中不会复制 resource fork 和 Finder info。

4.4.3 怎样读取（或写入）二进制数据？

要读写复杂的二进制数据格式，最好使用 `struct` 模块。该模块可以读取包含二进制数据（通常是数字）的字符串并转换为 Python 对象，反之亦然。

举例来说，下面的代码会从文件中以大端序格式读取一个 2 字节的整型和一个 4 字节的整型：

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

格式字符串中的‘>’强制以大端序读取数据；字母‘h’从字符串中读取一个“短整型”（2 字节），字母‘l’读取一个“长整型”（4 字节）。

对于更常规的数据（例如整型或浮点类型的列表），你也可以使用 `array` 模块。

備🄵： 要读写二进制数据的话，需要强制以二进制模式打开文件（这里为 `open()` 函数传入 `"rb"`）。如果（默认）传入 `"r"` 的话，文件会以文本模式打开，`f.read()` 会返回 `str` 对象，而不是 `bytes` 对象。

4.4.4 似乎 os.popen() 创建的管道不能使用 os.read()，这是为什么？

`os.read()` 是一个底层函数，它接收的是文件描述符——用小整型数表示的打开的文件。`os.popen()` 创建的是一个高级文件对象，和内建的 `open()` 方法返回的类型一样。因此，如果要从 `os.popen()` 创建的管道 *p* 中读取 *n* 个字节的话，你应该使用 `p.read(n)`。

4.4.5 怎样访问（RS232）串口？

对于 Win32，POSIX（Linux，BSD 等），Jython：

> http://pyserial.sourceforge.net

对于 Unix，查看 Mitch Chapman 发布的帖子：

> https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

4.4.6 为什么关闭 sys.stdout（stdin，stderr）并不会真正关掉它？

Python 文件对象 是一个对底层 C 文件描述符的高层抽象。

对于在 Python 中通过内建的 `open()` 函数创建的多数文件对象来说，`f.close()` 从 Python 的角度将其标记为已关闭，并且会关闭底层的 C 文件描述符。在 `f` 被垃圾回收的时候，析构函数中也会自动处理。

但由于 stdin，stdout 和 stderr 在 C 中的特殊地位，在 Python 中也会对它们做特殊处理。运行 `sys.stdout.close()` 会将 Python 的文件对象标记为已关闭，但是 * 不会 * 关闭与之关联的文件描述符。

要关闭这三者的 C 文件描述符的话，首先你应该确认确实需要关闭它（比如，这可能会影响到处理 I/O 的拓展）。如果确实需要这么做的话，使用 `os.close()`：

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

或者也可以使用常量 0，1，2 代替。

4.5 网络 / Internet 编程

4.5.1 Python 中的 WWW 工具是什么？

参阅代码库参考手册中 internet 和 netdata 这两章的内容。Python 有大量模块来帮助你构建服务端和客户端 web 系统。

Paul Boddie 维护了一份可用框架的概览，见 https://wiki.python.org/moin/WebProgramming 。

Cameron Laird 维护了一份关于 Python web 技术的实用网页的集合，见 http://phaseit.net/claird/comp.lang.python/web_python 。

4.5.2 怎样模拟发送 CGI 表单（METHOD=POST）？

我需要通过 POST 表单获取网页，有什么代码能简单做到吗?

是的。这里是一个使用 `urllib.request` 的简单例子:

```
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                             '/cgi-bin/some-cgi-script', data=qs)
with req:
    msg, hdrs = req.read(), req.info()
```

注意，通常在百分号编码的 POST 操作中，查询字符串必须使用 `urllib.parse.urlencode()` 处理一下。举个例子，如果要发送 `name=Guy Steele, Jr.` 的话:

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

也参考:

查看 urllib-howto 获取更多示例。

4.5.3 生成 HTML 需要使用什么模块？

你可以在 Web 编程 wiki 页面 找到许多有用的链接。

4.5.4 怎样使用 Python 脚本发送邮件？

使用 smtplib 标准库模块。

下面是一个很简单的交互式发送邮件的代码。这个方法适用于任何支持 SMTP 协议的主机。

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

在 Unix 系统中还可以使用 sendmail。sendmail 程序的位置在不同系统中不一样，有时是在 /usr/lib/sendmail，有时是在 /usr/sbin/sendmail。sendmail 手册页面会对你有所帮助。以下是示例代码：

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.5 socket 的 connect() 方法怎样避免阻塞？

通常会用 select 模块处理 socket 异步 I/O。

要防止 TCP 连接发生阻塞，你可以将 socket 设为非阻塞模式。这样当你执行 socket.connect() 时，你将或是立即完成连接（不大可能）或是收到一个包含 .errno 错误码的异常。errno.EINPROGRESS 表示连接正在进行但还没有完成。不同的操作系统将返回不同的值，因此你需要确认你的系统会返回什么值。

你可以使用 socket.connect_ex() 方法来避免生成异常。它将只返回 errno 值。要进行轮询，你可以稍后再次调用 socket.connect_ex() -- 0 或 errno.EISCONN 表示连接已完成 -- 或者你也可以将此 socket 传给 select.select() 来检查它是否可写。

備🄴: asyncio 模块提供了通用的单线程并发异步库，它可被用来编写非阻塞的网络代码。第三方的 Twisted 库是一个热门且功能丰富的替代选择。

4.6 数据库

4.6.1 Python 中有数据库包的接口吗？

有的

标准 Python 还包含了基于磁盘的哈希接口例如 `DBM` 和 `GDBM`。除此之外还有 `sqlite3` 模块，该模块提供了一个轻量级的基于磁盘的关系型数据库。

大多数关系型数据库都已经支持。查看 数据库编程 wiki 页面 获取更多信息。

4.6.2 在 Python 中如何实现持久化对象？

`pickle` 库模块以一种非常通用的方式解决了这个问题（虽然你依然不能用它保存打开的文件、套接字或窗口之类的东西），此外 `shelve` 库模块可使用 pickle 和 (g)dbm 来创建包含任意 Python 对象的持久化映射。

4.7 数学和数字

4.7.1 Python 中怎样生成随机数？

`random` 标准库模块实现了随机数生成器，使用起来非常简单：

```
import random
random.random()
```

这个函数会返回 [0, 1) 之间的随机浮点数。

该模块中还有许多其他的专门的生成器，例如：

- `randrange(a, b)` 返回 [a, b) 区间内的一个整型数。
- `uniform(a, b)` 返回 [a, b) 区间之间的浮点数。
- `normalvariate(mean, sdev)` 使用正态（高斯）分布采样。

还有一些高级函数直接对序列进行操作，例如：

- `choice(S)` 从给定的序列中随机选择一个元素。
- `shuffle(L)` 会对列表执行原地重排，即将其随机地打乱。

还有 `Random` 类，你可以将其实例化，用来创建多个独立的随机数生成器。

CHAPTER 5

扩展/嵌入常见问题

5.1 可以使用 C 语言创建自己的函数吗？

是的，您可以在 C 中创建包含函数、变量、异常甚至新类型的内置模块。在文档 extending-index 中有说明。

大多数中级或高级的 Python 书籍也涵盖这个主题。

5.2 可以使用 C++ 语言创建自己的函数吗？

是的，可以使用 C ++ 中兼容 C 的功能。在 Python include 文件周围放置 ' *extern* "*C*" *{...}* '，并在 Python 解释器调用的每个函数之前放置 `extern "C"`。具有构造函数的全局或静态 C ++ 对象可能不是一个好主意。

5.3 C 很难写，有没有其他选择？

编写自己的 C 扩展有很多选择，具体取决于您要做的事情。

Cython 及其相关的 Pyrex 是接受稍微修改过的 Python 形式并生成相应 C 代码的编译器。Cython 和 Pyrex 可以编写扩展而无需学习 Python 的 C API。

如果需要连接到某些当前不存在 Python 扩展的 C 或 C ++ 库，可以尝试使用 SWIG 等工具包装库的数据类型和函数。SIP ， CXX Boost , 或 Weave 也是包装 C ++ 库的替代方案。

5.4 如何在 C 中执行任意 Python 语句？

执行此操作的最高层级函数为 PyRun_SimpleString()，它接受单个字符串参数用于在模块 __main__ 的上下文中执行并在成功时返回 0 而在发生异常 (包括 SyntaxError) 时返回 -1。如果你想要更多可控性，可以使用 PyRun_String()；请在 Python/pythonrun.c 中查看 PyRun_SimpleString() 的源码。

5.5 如何在 C 中对任意 Python 表达式求值？

可以调用前一问题中介绍的函数 PyRun_String() 并附带起始标记符 Py_eval_input；它会解析表达式，对其求值并返回结果值。

5.6 如何从 Python 对象中提取 C 的值？

这取决于对象的类型。如果是元组，PyTuple_Size() 可返回其长度而 PyTuple_GetItem() 可返回指定序号上的项。对于列表也有类似的函数 PyListSize() 和 PyList_GetItem()。

对于字节串，PyBytes_Size() 可返回其长度而 PyBytes_AsStringAndSize() 提供一个指向其值和长度的指针。请注意 Python 字节串可能为空，因此 C 的 strlen() 不应被使用。

要检测一个对象的类型，首先要确保它不为 NULL，然后使用 PyBytes_Check(), PyTuple_Check(), PyList_Check() 等等。

还有一个针对 Python 对象的高层级 API，通过所谓的‘抽象’接口提供——请参阅 Include/abstract.h 了解详情。它允许使用 PySequence_Length(), PySequence_GetItem() 这样的调用来与任意种类的 Python 序列进行对接，此外还可使用许多其他有用的协议例如数字 (PyNumber_Index() 等) 以及 PyMapping API 中的各种映射等等。

5.7 如何使用 Py_BuildValue() 创建任意长度的元组？

不可以。应该使用 PyTuple_Pack()。

5.8 如何从 C 调用对象的方法？

可以使用 PyObject_CallMethod() 函数来调用某个对象的任意方法。形参为该对象、要调用的方法名、类似 Py_BuildValue() 所用的格式字符串以及要传给方法的参数值:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

这适用于任何具有方法的对象——不论是内置方法还是用户自定义方法。你需要负责对返回值进行最终的 Py_DECREF() 处理。

例如调用某个文件对象的”seek” 方法并传入参数 10, 0 (假定文件对象的指针为”f”):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

请注意由于 PyObject_CallObject() 总是接受一个元组作为参数列表，要调用不带参数的函数，则传入格式为"()"，要调用只带一个参数的函数，则应将参数包含于圆括号中，例如"(i)"。

5.9 如何捕获 PyErr_Print()（或打印到 stdout / stderr 的任何内容）的输出？

在 Python 代码中，定义一个支持 write() 方法的对象。将此对象赋值给 sys.stdout 和 sys.stderr。调用 print_error 或者只是允许标准回溯机制生效。在此之后，输出将转往你的 write() 方法所指向的任何地方。

做到这一点的最简单方式是使用 io.StringIO 类：

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

实现同样效果的自定义对象看起来是这样的：

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

5.10 如何从 C 访问用 Python 编写的模块？

你可以通过如下方式获得一个指向模块对象的指针：

```
module = PyImport_ImportModule("<modulename>");
```

如果模块尚未被导入（即它还不存在于 sys.modules 中），这会初始化该模块；否则它只是简单地返回 sys.modules["<modulename>"] 的值。请注意它并不会将模块加入任何命名空间——它只是确保模块被初始化并存在于 sys.modules 中。

之后你就可以通过如下方式来访问模块的属性（即模块中定义的任何名称）：

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

调用 PyObject_SetAttrString() 为模块中的变量赋值也是可以的。

5.11 如何在 Python 中对接 C ++ 对象？

根据你的需求，可以选择许多方式。手动的实现方式请查阅 " 扩展与嵌入" 文档来入门。需要知道的是对于 Python 运行时系统来说，C 和 C++ 并不没有太大的区别——因此围绕一个 C 结构（指针）类型构建新 Python 对象的策略同样适用于 C++ 对象。

有关 C ++ 库，请参阅*C 很难写，有没有其他选择？*

5.12 我使用 Setup 文件添加了一个模块，为什么 make 失败了？

安装程序必须以换行符结束，如果没有换行符，则构建过程将失败。（修复这个需要一些丑陋的 shell 脚本编程，而且这个 bug 很小，看起来不值得花这么大力气。）

5.13 如何调试扩展？

将 GDB 与动态加载的扩展名一起使用时，在加载扩展名之前，不能在扩展名中设置断点。

在您的 `.gdbinit` 文件中（或交互式）添加命令：

```
br _PyImport_LoadDynamicModule
```

然后运行 GDB：

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 我想在 Linux 系统上编译一个 Python 模块，但是缺少一些文件。为什么？

大多数打包的 Python 版本不包含 `/usr/lib/python2.x/config/` 目录，该目录中包含编译 Python 扩展所需的各种文件。

对于 Red Hat，安装 python-devel RPM 以获取必要的文件。

对于 Debian，运行 `apt-get install python-dev` 。

5.15 如何区分 "输入不完整" 和 "输入无效"？

有时，希望模仿 Python 交互式解释器的行为，在输入不完整时 (例如，您键入了 "if" 语句的开头，或者没有关闭括号或三个字符串引号)，给出一个延续提示，但当输入无效时，立即给出一条语法错误消息。

在 Python 中，您可以使用 `codeop` 模块，该模块非常接近解析器的行为。例如，IDLE 就使用了这个。

在 C 中执行此操作的最简单方法是调用 `PyRun_InteractiveLoop()` （可能在单独的线程中）并让 Python 解释器为您处理输入。您还可以设置 `PyOS_ReadlineFunctionPointer()` 指向您的自定义输入函数。有关更多提示，请参阅 `Modules/readline.c` 和 `Parser/myreadline.c` 。

但是，有时必须在与其他应用程序相同的线程中运行嵌入式 Python 解释器，并且不能允许 `PyRun_InteractiveLoop()` 在等待用户输入时停止。那么另一个解决方案是调用

PyParser_ParseString() 并测试 e.error 等于 E_EOF ，如果等于，就意味着输入不完整。这是一个示例代码片段，未经测试，灵感来自 Alex Farber 的代码:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
  /* code should end in \n */
  /* return -1 for error, 0 for incomplete, 1 for complete */
{
  node *n;
  perrdetail e;

  n = PyParser_ParseString(code, &_PyParser_Grammar,
                           Py_file_input, &e);
  if (n == NULL) {
    if (e.error == E_EOF)
      return 0;
    return -1;
  }

  PyNode_Free(n);
  return 1;
}
```

另一个解决方案是尝试使用 Py_CompileString() 编译接收到的字符串。如果编译时没有出现错误，请尝试通过调用 PyEval_EvalCode() 来执行返回的代码对象。否则，请将输入保存到以后。如果编译失败，找出是错误还是只需要更多的输入-从异常元组中提取消息字符串，并将其与字符串“分析时意外的 EOF”进行比较。下面是使用 GNUreadline 库的完整示例 (您可能希望在调用 readline() 时忽略 **SIGINT**):

```
#include <stdio.h>
#include <readline.h>

#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
  int i, j, done = 0;                           /* lengths of line, code */
  char ps1[] = ">>> ";
  char ps2[] = "... ";
  char *prompt = ps1;
  char *msg, *line, *code = NULL;
  PyObject *src, *glb, *loc;
  PyObject *exc, *val, *trb, *obj, *dum;

  Py_Initialize ();
  loc = PyDict_New ();
  glb = PyDict_New ();
  PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

  while (!done)
  {
```

(下页继续)

(繼續上一頁)

```
    line = readline (prompt);

    if (NULL == line)                           /* Ctrl-D pressed */
    {
      done = 1;
    }
    else
    {
      i = strlen (line);

      if (i > 0)
        add_history (line);                     /* save non-empty lines */

      if (NULL == code)                         /* nothing in code yet */
        j = 0;
      else
        j = strlen (code);

      code = realloc (code, i + j + 2);
      if (NULL == code)                         /* out of memory */
        exit (1);

      if (0 == j)                               /* code was empty, so */
        code[0] = '\0';                         /* keep strncat happy */

      strncat (code, line, i);                  /* append line to code */
      code[i + j] = '\n';                       /* append '\n' to code */
      code[i + j + 1] = '\0';

      src = Py_CompileString (code, "<stdin>", Py_single_input);

      if (NULL != src)                          /* compiled just fine - */
      {
        if (ps1  == prompt ||                   /* ">>> " or */
            '\n' == code[i + j - 1])            /* "... " and double '\n' */
        {                                               /* so execute it */
          dum = PyEval_EvalCode (src, glb, loc);
          Py_XDECREF (dum);
          Py_XDECREF (src);
          free (code);
          code = NULL;
          if (PyErr_Occurred ())
            PyErr_Print ();
          prompt = ps1;
        }
      }                                         /* syntax error or E_EOF? */
      else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
      {
        PyErr_Fetch (&exc, &val, &trb);         /* clears exception! */

        if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
        {
          Py_XDECREF (exc);
          Py_XDECREF (val);
          Py_XDECREF (trb);
          prompt = ps2;
        }
        else                                    /* some other syntax error */
        {
          PyErr_Restore (exc, val, trb);
```

(下页继续)

(繼續上一頁)

```
          PyErr_Print ();
          free (code);
          code = NULL;
          prompt = ps1;
        }
      }
      else                                    /* some non-syntax error */
      {
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
      }

      free (line);
    }
  }

  Py_XDECREF(glb);
  Py_XDECREF(loc);
  Py_Finalize();
  exit(0);
}
```

5.16 如何找到未定义的 g++ 符号 __builtin_new 或 __pure_virtual？

要动态加载 g ++ 扩展模块，必须重新编译 Python，要使用 g ++ 重新链接（在 Python Modules Makefile 中更改 LINKCC），及链接扩展模块（例如：`g++ -shared -o mymodule.so mymodule.o`）。

5.17 能否创建一个对象类，其中部分方法在 C 中实现，而其他方法在 Python 中实现（例如通过继承）？

是的，您可以继承内置类，例如 `int`，`list`，`dict` 等。

Boost Python 库（BPL，http：//www.boost.org/libs/python/doc/index.html）提供了一种从 C ++ 执行此操作的方法（即，您可以使用 BPL 继承自 C ++ 编写的扩展类）。

CHAPTER 6

FAQ：在 Windows 使用 Python

6.1 在 Windows 作業系統🄵我想執行 Python 程式，要怎🄵做？

這個問題的答案可能有點🄴雜。如果你經常使用「命令提示字元」執行程式，那這對你來🄵不會是什🄴難事。如果不然，那就需要更仔細的🄴明了。

除非你在使用某种集成开发环境，否则你将会在被称为“DOS 窗口”或“命令提示符窗口”的地方输入 Windows 命令。通常你可以在搜索栏搜索 cmd 来创建这种窗口。你应该能够识别你何时打开了这样的窗口，因为你将看到一个 Windows“命令提示符”，通常看起来是这样：

```
C:\>
```

前面的字母可能会不同，而且后面有可能会有其他东西，所以你也许会看到类似这样的东西：

```
D:\YourName\Projects\Python>
```

出现的内容具体取决与你的电脑如何设置和你最近用它做的事。当你启动了这样一个窗口后，就可以开始运行 Python 程序了。

Python 脚本需要被另外一个叫做 Python 解释器的程序来处理。解释器读取脚本，把它编译成字节码，然后执行字节码来运行你的程序。所以怎样安排解释器来处理你的 Python 脚本呢？

首先，确保命令窗口能够将“py”识别为指令来开启解释器。如果你打开过一个命令窗口，尝试输入命令 py 然后按回车：

```
C:\Users\YourName> py
```

然后你应当看见类似类似这样的东西：

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]␣
↪on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

解释器已经以“交互模式”打开。这意味着你可以交互输入 Python 语句或表达式，并在等待时执行或评估它们。这是 Python 最强大的功能之一。输入几个表达式并看看结果：

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

许多人把交互模式当作方便和高度可编程的计算器。想结束交互式 Python 会话时，调用 `exit()` 函数，或者按住 `Ctrl` 键时输入 `Z`，之后按 `Enter` 键返回 Windows 命令提示符。

你可能发现在开始菜单有这样一个条目 *开始 ▸ 所有程序 ▸ Python 3.x ▸ Python (命令行)*，运行它后会出现一个有着 >>> 提示的新窗口。在此之后，如果调用 `exit()` 函数或按 `Ctrl-Z` 组合键后窗口将会消失。Windows 会在这个窗口中运行一个 “python” 命令，并且在你终止解释器的时候关闭它。

现在我们知道 `py` 命令已经被识别，可以输入 Python 脚本了。你需要提供 Python 脚本的绝对路径或相对路径。假设 Python 脚本位于桌面上并命名为 `hello.py`，并且命令提示符在用户主目录打开，那么可以看到类似于这样的东西:

```
C:\Users\YourName>
```

那么现在可以让 “py“命令执行你的脚本，只需要输入 “py“ 和脚本路径:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 我怎么让 Python 脚本可执行？

在 Windows 上，标准 Python 安装程序已将.py 扩展名与文件类型 (Python.File) 相关联，并为该文件类型提供运行解释器的打开命令 (`D:\Program Files\Python\python.exe "%1" %*`)。这足以使脚本在命令提示符下作为 “foo.py” 命令被执行。如果希望通过简单地键入 “foo” 而无需输入文件扩展名来执行脚本，则需要将.py 添加到 PATHEXT 环境变量中。

6.3 为什么有时候 Python 程序会启动缓慢？

通常，Python 在 Windows 上启动得很快，但偶尔会有错误报告说 Python 突然需要很长时间才能启动。更令人费解的是，在其他配置相同的 Windows 系统上，Python 却可以工作得很好。

该问题可能是由于计算机上的杀毒软件配置错误造成的。当将病毒扫描配置为监视文件系统中所有读取行为时，一些杀毒扫描程序会导致两个数量级的启动开销。请检查你系统安装的杀毒扫描程序的配置，确保两台机它们是同样的配置。已知的，McAfee 杀毒软件在将它设置为扫描所有文件系统访问时，会产生这个问题。

6.4 我怎样使用 Python 脚本制作可执行文件？

请参阅*如何由 Python 脚本创建能独立运行的二进制程序？* 查看可用来生成可执行文件的工具清单。

6.5 *.pyd 文件和 DLL 文件相同吗？

是的，.pyd 文件也是 dll，但有一些差异。如果你有一个名为 `foo.pyd` 的 DLL，那么它必须有一个函数 `PyInit_foo()`。然后你可以编写 Python 代码“import foo”，Python 将搜索 foo.pyd（以及 foo.py、foo.pyc）。如果找到它，将尝试调用 `PyInit_foo()` 来初始化它。你不应将.exe 与 foo.lib 链接，因为这会导致 Windows 要求存在 DLL。

请注意，foo.pyd 的搜索路径是 PYTHONPATH，与 Windows 用于搜索 foo.dll 的路径不同。此外，foo.pyd 不需要存在来运行你的程序，而如果你将程序与 dll 链接，则需要 dll。当然，如果你想 `import foo`，则需要 foo.pyd。在 DLL 中，链接在源代码中用 `__declspec(dllexport)` 声明。在.pyd 中，链接在可用函数列表中定义。

6.6 我怎样将 Python 嵌入一个 Windows 程序？

在 Windows 应用程序中嵌入 Python 解释器可以总结如下：

1. 请 _ 不要 _ 直接在你的.exe 文件中内置 Python。在 Windows 上，Python 必须是一个 DLL，这样才可以处理导入的本身就是 DLL 的模块。（这是第一个未记录的关键事实。）相反，链接到 `python`*NN*`.dll`；它通常安装在 `C:\Windows\System` 中。*NN* 是 Python 版本，如数字“33”代表 Python 3.3。

   你可以通过两种不同的方式链接到 Python。加载时链接意味着链接到 `python`*NN*`.lib`，而运行时链接意味着链接 `python`*NN*`.dll`。（一般说明：`python` *NN*`.lib` 是所谓的“import lib”，对应于 `python`*NN*`.dll`。它只定义了链接器的符号。）

   运行时链接极大地简化了链接选项，一切都在运行时发生。你的代码必须使用 Windows 的 `LoadLibraryEx()` 程序加载 `python`*NN*`.dll`。代码还必须使用使用 Windows 的 `GetProcAddress()` 例程获得的指针访问 `python`*NN*`.dll` 中程序和数据（即 Python 的 C API）。宏可以使这些指针对任何调用 Python C API 中的例程的 C 代码都是透明的。

   Borland 提示：首先使用 Coff2Omf.exe 将 `python`*NN*`.lib` 转换为 OMF 格式。

2. 如果你使用 SWIG，很容易创建一个 Python“扩展模块”，它将使应用程序的数据和方法可供 Python 使用。SWIG 将为你处理所有蹩脚的细节。结果是你将链接到.exe 文件中的 C 代码 (!) 你不必创建 DLL 文件，这也简化了链接。

3. SWIG 将创建一个 init 函数（一个 C 函数），其名称取决于扩展模块的名称。例如，如果模块的名称是 leo，则 init 函数将被称为 initleo()。如果您使用 SWIG 阴影类，则 init 函数将被称为 initleoc()。这初始化了一个由阴影类使用的隐藏辅助类。

   你可以将步骤 2 中的 C 代码链接到.exe 文件的原因是调用初始化函数等同于将模块导入 Python！（这是第二个关键的未记载事实。）

4. 简而言之，你可以用以下代码使用扩展模块初始化 Python 解释器。

```
#include "python.h"
...
Py_Initialize();  // Initialize Python.
initmyAppc();  // Initialize (import) the helper class.
PyRun_SimpleString("import myApp");  // Import the shadow class.
```

5. Python C API 存在两个问题，如果你使用除 MSVC 之外的编译器用于构建 python.dll，这将会变得明显。

   问题 1：采用 FILE* 参数的所谓“极高级”函数在多编译器环境中不起作用，因为每个编译器的 FILE 结构体概念都不同。从实现的角度来看，这些是非常 _ 低 _ 级的功能。

   问题 2：在为 void 函数生成包装器时，SWIG 会生成以下代码：

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Py_None 是一个宏，它扩展为对 pythonNN.dll 中名为 _Py_NoneStruct 的复杂数据结构的引用。同样，此代码将在多编译器环境中失败。将此类代码替换为：

```
return Py_BuildValue("");
```

有可能使用 SWIG 的 `%typemap` 命令自动进行更改，但我无法使其工作（我是一个完全的 SWIG 新手）。

6. 使用 Python shell 脚本从 Windows 应用程序内部建立 Python 解释器窗口并不是一个好主意；生成的窗口将独立于应用程序的窗口系统。相反，你（或 wxPythonWindow 类）应该创建一个“本机”解释器窗口。将该窗口连接到 Python 解释器很容易。你可以将 Python 的 i/o 重定向到支持读写的 _ 任意 _ 对象，因此你只需要一个包含 read() 和 write() 方法的 Python 对象（在扩展模块中定义）。

6.7 如何让编辑器不要在我的 Python 源代码中插入 tab ？

本 FAQ 不建议使用制表符，Python 样式指南 **PEP 8** ，为发行的 Python 代码推荐 4 个空格；这也是 Emacs python-mode 默认值。

在任何编辑器下，混合制表符和空格都是一个坏主意。MSVC 在这方面没有什么不同，并且很容易配置为使用空格：点击 *Tools* ▸ *Options* ▸ *Tabs*，对于文件类型“Default”，设置“Tab size”和“Indent size”为 4，并选择“插入空格”单选按钮。

如果混合制表符和空格导致前导空格出现问题，Python 会引发 `IndentationError` 或 `TabError` 。你还可以运行 `tabnanny` 模块以批处理模式检查目录树。

6.8 如何在不阻塞的情况下检查按键？

使用 `msvcrt` 模块。这是一个标准的 Windows 专属扩展模块。它定义了一个函数 `kbhit()` 用于检查是否有键盘中的某个键被按下，以及 `getch()` 用于获取一个字符而不将其回显。

CHAPTER 7

圖形化使用者界面常見問答集

7.1 常見圖形化使用者界面 (GUI) 問題

7.2 Python 有哪些 GUI 工具包？

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called tkinter. This is probably the easiest to install (since it comes included with most binary distributions of Python) and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page. Tcl/Tk is fully portable to the macOS, Windows, and Unix platforms.

存在多种选项，具体取决于你的目标平台。Python Wiki 上提供了一个 跨平台 和 平台专属 的 GUI 框架列表。

7.3 有关 Tkinter 的问题

7.3.1 我怎样“冻结”Tkinter 程序？

Freeze 是一个用来创建独立应用程序的工具。当冻结 (freeze) Tkinter 程序时，程序并不是真的能够独立运行，因为程序仍然需要 Tcl 和 Tk 库。

一种解决方法是将程序与 Tcl 和 Tk 库一同发布，并且在运行时使用环境变量 `TCL_LIBRARY` 和 `TK_LIBRARY` 指向他们的位置。

为了获得真正能独立运行的应用程序，来自库里的 Tcl 脚本也需要被整合进应用程序。一个做这种事情的工具叫 SAM (stand-alone modules，独立模块)，它是 Tix distribution (http://tix.sourceforge.net/) 的一部分。

在启用 SAM 时编译 Tix，在 Python 文件 `Modules/tkappinit.c` 中执行对 `Tclsam_init()` 等的适当调用，并且将程序与 libtclsam 和 libtksam 相链接（可能也要包括 Tix 的库）。

7.3.2 在等待 I/O 操作时能够处理 Tk 事件吗？

在 Windows 以外的其他平台上可以，你甚至不需要使用线程！但是你必须稍微修改一下你的 I/O 代码。Tk 有与 Xt 的 `XtAddInput()` 对应的调用，它允许你注册一个回调函数，当一个文件描述符可以进行 I/O 操作的时候，Tk 主循环将会调用这个回调函数。参见 tkinter-file-handlers。

7.3.3 在 Tkinter 中键绑定不工作：为什么？

经常听到的抱怨是：已经通过 `bind()` 方法绑定了事件的处理程序，但是，当按下相关的按键后，这个处理程序却没有执行。

最常见的原因是，那个绑定的控件没有“键盘焦点”。请在 Tk 文档中查找 focus 指令。通常一个控件要获得“键盘焦点”，需要点击那个控件（而不是标签；请查看 takefocus 选项）。

CHAPTER 8

什麼 Python 被安裝在我的機器上? 常見問答集

8.1 什麼是 Python?

Python 是一種程式語言。它被使用於不同種類的應用程式中。因為 Python 屬於容易學習的語言，它被使用在一些高中和學院中作為介紹程式語言的工具；但它也被專業的軟體開發人員所使用，例如：Google，美國太空總署與盧卡斯電影公司。

若你希望學習更多關於 Python 語言，可以從"Python 初學者指引" 網站 <https://wiki.python.org/moin/BeginnersGuide>‘ 進行閱讀。

8.2 為什麼 Python 被安裝在我的機器上?

若你發現曾安裝 Python 於系統中，但不記得何時安裝過，以下有幾種可能的方法可以得知。

- 也許其他使用此電腦的使用者希望學習撰寫程式且安裝 Python；你必須指出誰曾經使用此機器且可能安裝過。
- 第三方應用程式安裝於機器中可能以 Python 語言撰寫且安裝 Python。有許多類似的應用程式，從 GUI 程式到網路伺服器和管理者本。
- 一些安裝 Windows 的機器也被安裝 Python。截至目前此文件交付，我們得知 HP 出廠的機器預設安裝 Python。顯然的 HP 管理工具程式是透過 Python 語言所撰寫。
- 許多相容於 Unix 系統，例如：maxOS 和一些 Linux 發行版本預設安裝 Python；安裝時被包含在基本安裝功能。

8.3 我能F自行F除 Python 嗎?

需要依據 Python 的安裝方式F定。

若有人不小心安裝了 Python，可自行移除移除它。Windows 作業系統中，請於控制台中尋找新增/移除程式進行反安裝。

若 Python 是透過第三方元件應用程式被安裝時，也可自行移除，不過請小心該應用程式將無法正常執行。你應該使用應用程式反安裝功能而非自行F除 Python 目F

當作業系統預設安裝 Python，不建議移除它。對你而言某些工具程式是重要不可或缺的，若自行移除它，透過 Python 撰寫的工具程式將無法正常執行。重新安裝整個系統仍需再次處理這些事情。

APPENDIX A

術語表

>>> 互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

... 可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）🄵部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，互動式 shell 顯示的預設 Python 提示字元。
- 🄵建常數 `Ellipsis`。

2to3 一個試著將 Python 2.x 程式碼轉🄵🄵 Python 3.x 程式碼的工具，它是透過處理大部分的不相容性來達成此目的，而這些不相容性能🄵透過剖析原始碼和遍歷剖析樹而被檢測出來。

2to3 在可以標准函式庫中以 `lib2to3` 被使用；它提供了一個獨立的入口點，在 `Tools/scripts/2to3`。請參🄵 2to3-reference。

abstract base class（抽象基底類🄵） 抽象基底類🄵（又稱🄵 ABC）提供了一種定義介面的方法，作🄵*duck-typing*（鴨子型🄵）的補充。其他類似的技術，像是 `hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC 🄵用🄵擬的 subclass（子類🄵），它們🄵不繼承自另一個 class（類🄵），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參🄵 `abc` 模組的🄵明文件。Python 有許多🄵建的 ABC，用於資料結構（在 `collections.abc` 模組）、數字（在 `numbers` 模組）、串流（在 `io` 模組）及 import 尋檢器和載入器（在 `importlib.abc` 模組）。你可以使用 `abc` 模組建立自己的 ABC。

annotation（🄵釋） 一個與變數、class 屬性、函式的參數或回傳值相關聯的標🄵。照慣例，它被用來作🄵*type hint*（型🄵提示）。

在運行時 (runtime)，區域變數的🄵釋無法被存取，但全域變數、class 屬性和函式的🄵解，會分🄵被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參🄵*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的🄵明。

argument（引數） 呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- *關鍵字引數 (keyword argument)*：在函式呼叫中，以識🄵字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）🄵的值被傳遞的引數。例如，`3` 和 `5` 都是以下 `complex()` 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引數 *(positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作🄵 `*` 之後的*iterable*（可🄵代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

```
complex(3, 5)
complex(*(3, 5))
```

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參🄵 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參🄵術語表的*parameter*（參數）條目、常見問題中的*引數和參數之間的差*🄵，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器） 一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步🄵生器） 一個會回傳*asynchronous generator iterator*（非同步🄵生器🄵代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 🄵圈的值。

這個術語通常用來表示一個非同步🄵生器函式，但在某些情境中，也可能是表示非同步🄵生器🄵代器 *(asynchronous generator iterator)*。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

一個非同步🄵生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步🄵生器🄵代器） 一個由*asynchronous generator*（非同步🄵生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步🄵代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步🄵生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，🄵記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步🄵生器🄵代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參🄵 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可🄵代物件） 一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步🄵代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步🄵代器） 一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步🄵代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attribute（屬性） 一個與某物件相關聯的值，該值能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

awaitable（可等待物件） 一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參🄵 **PEP 492**。

BDFL Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進制檔案） 一個能🄵讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進制檔案的例子有：以二進制模式（`'rb'`、`'wb'` 或 `'rb+'`）開🄵的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參🄵*text file*（文字檔案），它是一個能🄵讀取和寫入 `str` 物件的檔案物件。

bytes-like object（類位元組串物件） 一個支援 bufferobjects 且能🄵匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進制資料的各種運算；這些運算包括壓縮、儲存至二進制檔案和透過 socket（插座）發送。

有些運算需要二進制資料是可變的。🄵明文件通常會將這些物件稱🄵「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進

制資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 bytes，以及 bytes 物件的 memoryview。

bytecode（位元組碼） Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的🄵部表示法。該位元組碼也會被暫存在 .pyc 檔案中，以便第二次執行同一個檔案時能🄵更快速（可以不用從原始碼重新編譯🄵位元組碼）。這種「中間語言 (intermediate language)」據🄵是運行在一個*virtual machine*（🄵擬機器）上，該🄵擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 🄵擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的🄵明文件中找到。

callback（回呼） 作🄵引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類🄵） 一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類🄵變數） 一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

coercion（🄵制轉型） 在涉及兩個不同型🄵引數的操作過程中，將某一種型🄵的實例🄵🄵另一種型🄵的隱式轉🄵 (implicit conversion) 過程。例如，int(3.15) 會將浮點數轉🄵🄵整數 3，但在 3+4.5 中，每個引數是不同的型🄵（一個 int，一個 float），而這兩個引數必須在被轉🄵🄵相同的型🄵之後才能相加，否則就會引發 TypeError。如果🄵有🄵制轉型，即使所有的引數型🄵皆相容，它們都必須要由程式設計師正規化 (normalize) 🄵相同的值，例如，要用 float(3)+4.5 而不能只是 3+4.5。

complex number（🄵數） 一個我們熟悉的實數系統的擴充，在此所有數字都會被表示🄵一個實部和一個🄵部之和。🄵數就是🄵數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫🄵 i，在工程學中被寫🄵 j。Python 🄵建了對🄵數的支援，它是用後者的記法來表示🄵數；🄵部會帶著一個後綴的 j 被編寫，例如 3+1j。若要將 math 模組🄵的工具等效地用於🄵數，請使用 cmath 模組。🄵數的使用是一個相當進階的數學功能。如果你🄵有察覺到對它們的需求，那🄵幾乎能確定你可以安全地忽略它們。

context manager（情境管理器） 一個可以控制 with 陳述式中所見環境的物件，而它是透過定義 __enter__() 和 __exit__() method 來控制的。請參🄵 **PEP 343**。

context variable（情境變數） 一個變數，其值可以根據上下文的情境而有所不同。這類似執行緒區域儲存區 (Thread-Local Storage)，在其中，一個變數在每個執行緒可能具有不同的值。然而，關於情境變數，在一個執行緒中可能會有多個情境，而情境變數的主要用途，是在🄵行的非同步任務 (concurrent asynchronous task) 中，對於變數狀態的追🄵。請參🄵 contextvars。

contiguous（連續的） 如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視🄵是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程） 協程是副程式 (subroutine) 的一種更🄵廣義的形式。副程式是在某個時間點被進入🄵在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能🄵以 async def 陳述式被實作。另請參🄵 **PEP 492**。

coroutine function（協程函式） 一個回傳*coroutine*（協程）物件的函式。一個協程函式能以 async def 陳述式被定義，🄵可能會包含 await、async for 和 async with 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython Python 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器） 一個函式，它會回傳另一個函式，通常它會使用 @wrapper 語法，被應用🄵一種函式的變🄵 (function transformation)。裝飾器的常見範例是 classmethod() 和 staticmethod()。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

```
def f(...):
    ...
```

(下页继续)

(繼續上一頁)

```
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Class 也存在相同的概念，但在那F比較不常用。關於裝飾器的更多F容，請參F函式定義和 class 定義的F明文件。

descriptor（描述器） 任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行F會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或F除某個屬性時，會在 *a* 的 class 字典中查找名稱F *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因F它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、F態 method，以及對 super class（父類F）的參照。

關於描述器 method 的更多資訊，請參F descriptors 或描述器使用指南。

dictionary（字典） 一個關聯陣列 (associative array)，其中任意的鍵會被映射到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱F雜F (hash)。

dictionary comprehension（字典綜合運算） 一種緊密的方法，用來處理一個可F代物件中的全部或部分元素，F將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會F生一個字典，它包含了鍵 `n` 映射到值 `n ** 2`。請參F comprehensions。

dictionary view（字典檢視） 從 `dict.keys()`、`dict.values()` 及 `dict.items()` 回傳的物件被稱F字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要F制將字典檢視轉F完整的 list（串列），須使用 `list(dictview)`。請參F dict-views。

docstring（F明字串） 一個在 class、函式或模組中，作F第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，F被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於F明字串可以透過F省 (introspection) 來F覽，因此它是物件的F明文件存放的標准位置。

duck-typing（鴨子型F） 一種程式設計風格，它不是藉由檢查一個物件的型F來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一F鴨子而且叫起來像一F鴨子，那F它一定是一F鴨子。」）因FF調介面而非特定型F，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型F要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型F可以用抽象基底類F *(abstract base class)* 來補充。）然而，它通常會F用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

EAFP Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，F在該假設被推翻時再捕獲例外。這種乾F且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

expression（運算式） 一段可以被評估F求值的語法。F句話F，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，F非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組） 一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串） 以 `'f'` 或 `'F'` F前綴的字串文本通常被稱F「f 字串」，它是格式化的字串文本的縮寫。另請參F **PEP 498**。

file object（檔案物件） 一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能F協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標准輸入 / 輸出、記憶體F緩衝區、socket（插座）、管F (pipe) 等）的存取。檔案物件也被稱F類檔案物件 *(file-like object)* 或串流 *(stream)*。

實際上，有三種檔案物件：原始的二進制檔案、緩衝的二進制檔案和文字檔案。它們的介面在 `io` 模組中被定義。建立檔案物件的標准方法是使用 `open()` 函式。

file-like object（類檔案物件） *file object*（檔案物件）的同義字。

finder（尋檢器） 一個物件，它會嘗試F正在被 import 的模組尋找*loader*（載入器）。

從 Python 3.3 開始，有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參F **PEP 302**、**PEP 420** 和 **PEP 451** 以了解更多細節。

floor division（向下取整除法） 向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果F `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因F是 `-2.75` 被向下無條件舍去。請參F **PEP 238**。

function（函式） 一連串的陳述式，它能F向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參F*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式F釋） 函式參數或回傳值的一個*annotation*（F釋）。

函式F釋通常被使用於*型F提示*：例如，這個函式預期會得到兩個 `int` 引數，F會有一個 `int` 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函式F釋的語法在 function 章節有詳細解釋。

請參F*variable annotation* 和 **PEP 484**，皆有此功能的描述。

__future__ future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成F標准的語法或語義，來編譯當前的模組。而 `__future__` 模組則記F了 *feature*（功能）可能的值。透過 import 此模組F對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成F預設的功能：

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收） 當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能F檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（F生器） 一個會回傳*generator iterator*（F生器F代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能F生一系列的值，這些值可用於 for F圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個F生器函式，但在某些情境中，也可能是表示F生器F代器。萬一想表達的意思不F清楚，那就使用完整的術語，以避免歧義。

generator iterator（F生器F代器） 一個由*generator*（F生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，F記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當F生器F代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（F生器運算式） 一個會回傳F代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了F圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會F外層函數F生多個值：

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function（泛型函式） 一個由多個函式組成的函式，該函式會對不同的型F實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來F定。

另請參F*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型🄵） 一個能🄵被參數化 (parameterized) 的*type*（型🄵）；通常是一個容器，像是 `list`（串列）。它被用於型🄵提示和🄵釋。

請參🄵 **PEP 483** 了解更多細節，以及 `typing` 或 泛型🄵名型🄵 (generic alias type) 了解其用途。

GIL 請參🄵*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖） *CPython* 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的🄵建型🄵，如 `dict`）自動地避免🄵行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成🄵多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能🄵提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標准的或是第三方的，它們被設計成在執行壓縮或雜🄵等計算密集 (computationally-intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

過去對於建立「無限制執行緒」直譯器（以更高的精細度鎖定共享資料的直譯器）的努力🄵未成功，因🄵在一般的單一處理器情🄵下，效能會有所損失。一般認🄵，若要克服這個效能問題，會使實作變得🄵雜許多，進而付出更高的維護成本。

hash-based pyc（雜🄵架構的 pyc） 一個位元組碼 (bytecode) 暫存檔，它使用雜🄵值而不是對應原始檔案的最後修改時間，來確定其有效性。請參🄵 pyc-invalidation。

hashable（可雜🄵的） 如果一個物件有一個雜🄵值，該值在其生命🄵期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那🄵它就是一個可雜🄵物件。比較結果🄵相等的多個可雜🄵物件，它們必須擁有相同的雜🄵值。

可雜🄵性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因🄵這些資料結構都在其🄵部使用了雜🄵值。

大多數的 Python 不可變🄵建物件都是可雜🄵的；可變的容器（例如 list 或 dictionary）🄵不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜🄵的，它們本身才是可雜🄵的。若物件是使用者自定 class 的實例，則這些物件會被預設🄵可雜🄵的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜🄵值則是衍生自它們的 `id()`。

IDLE Python 的 Integrated Development Environment（整合開發環境）。IDLE 是一個基本的編輯器和直譯器環境，它和 Python 的標准發行版本一起被提供。

immutable（不可變物件） 一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要🄵定雜🄵值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（匯入路徑） 一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（匯入） 一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（匯入器） 一個能🄵尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的） Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們🄵且看到它們的結果。只要🄵動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常🄵大的方法（請記住 help(x)）。

interpreted（直譯的） Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因🄵有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯🄵期，不過它們的程式通常也運行得較慢。另請參🄵*interactive*（互動的）。

interpreter shutdown（直譯器關閉） 當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵🄵部結構。它也會多次呼叫*垃圾回收器 (garbage collector)*。這能🄵觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，🄵執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因🄵它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 __main__ 模組或正被運行的🄵本已經執行完成。

iterable（可🄵代物件） 一種能🄵一次回傳一個其中成員的物件。可🄵代物件的例子包括所有的序列型🄵（像是 list、str 和 tuple）和某些非序列型🄵，像是 dict、檔案物件，以及你所定義的任何 class 物件，只要那些 class 有 __iter__() method 或是實作*Sequence*（序列）語意的 __getitem__() method，該物件就是可🄵代物件。

可🄵代物件可用於 for 🄵圈和許多其他需要一個序列的地方 (zip()、map()...)。當一個可🄵代物件作🄵引數被傳遞給🄵建函式 iter() 時，它會🄵該物件回傳一個🄵代器。此🄵代器適用於針對一組值進行一遍 (one pass) 運算。使用🄵代器時，通常不一定要呼叫 iter() 或自行處理🄵代器物件。for 陳述式會自動地🄵你處理這些事，它會建立一個暫時性的未命名變數，用於在🄵圈期間保有該🄵代器。另請參🄵*iterator*（🄵代器）、*sequence*（序列）和*generator*（🄵生器）。

iterator（🄵代器） 一個表示資料流的物件。重🄵地呼叫🄵代器的 __next__() method（或是將它傳遞給🄵建函式 next()）會依序回傳資料流中的各項目。當不再有資料時，則會引發 StopIteration 例外。此時，該🄵代器物件已被用盡，而任何對其 __next__() method 的進一步呼叫，都只會再次引發 StopIteration。🄵代器必須有一個 __iter__() method，它會回傳🄵代器物件本身，所以每個🄵代器也都是可🄵代物件，且可以用於大多數適用其他可🄵代物件的場合。一個明顯的例外，是嘗試多遍🄵代 (multiple iteration passes) 的程式碼。一個容器物件（像是 list）在每次你將它傳遞給 iter() 函式或在 for 🄵圈中使用它時，都會🄵生一個全新的🄵代器。使用🄵代器嘗試此事（多遍🄵代）時，只會回傳在前一遍🄵代中被用過的、同一個已被用盡的🄵代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

key function（鍵函式） 鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，locale.strxfrm() 被用來🄵生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 min()、max()、sorted()、list.sort()、heapq.merge()、heapq.nsmallest()、heapq.nlargest() 和 itertools.groupby()。

有幾種方法可以建立一個鍵函式。例如，str.lower() method 可以作🄵不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 lambda 運算式被建造，例如 lambda r: (r[0], r[2])。另外，operator 模組提供了三個鍵函式的建構函式 (constructor)：attrgetter()、itemgetter() 和 methodcaller()。關於如何建立和使用鍵函式的範例，請參🄵如何排序。

keyword argument（關鍵字引數） 請參🄵*argument*（引數）。

lambda 由單一*expression*（運算式）所組成的一個匿名行🄵函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 lambda [parameters]: expression。

LBYL Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先🄵條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 if 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競🄵條件 (race condition) 的風險。例如以下程式碼 if key in mapping: return mapping[key]，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解🄵。

list（串列） 一個 Python 🄵建的*sequence*（序列）。🄵管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 O(1)。

list comprehension（串列綜合運算） 一種用來處理一個序列中的全部或部分元素，🄵將處理結果以一個 list 回傳的簡要方法。result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] 會🄵生一個字串 list，其中包含 0 到 255 範圍🄵，所有偶數的十六進位數 (0x..)。if 子句是選擇性的。如果省略它，則 range(256) 中的所有元素都會被處理。

loader（載入器） 一個能🄵載入模組的物件。它必須定義一個名🄵 load_module() 的 method（方法）。載入器通常是被*finder*（尋檢器）回傳。更多細節請參🄵 **PEP 302**，關於*abstract base class*（抽象基底類🄵），請參🄵 importlib.abc.Loader。

magic method（魔術方法） *special method*（特殊方法）的一個非正式同義詞。

mapping（對映） 一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類🄵）中，`Mapping` 或 `MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器） 一種經由搜尋 `sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參🄵 `importlib.abc.MetaPathFinder`。

metaclass（元類🄵） 一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類🄵）的列表。Metaclass 負責接受這三個引數，🄵建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特🄵之處在於它能🄵建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供🄵大且優雅的解🄵方案。它們已被用於記🄵屬性存取、增加執行緒安全性、追🄵物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法） 一個在 class 本體🄵被定義的函式。如果 method 作🄵其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成🄵它的第一個*argument*（引數）（此引數通常被稱🄵 `self`）。請參🄵*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序） 方法解析順序是在查找某個成員的過程中，base class（基底類🄵）被搜尋的順序。關於第 2.3 版至今，Python 直譯器所使用的演算法細節，請參🄵 Python 2.3 版方法解析順序。

module（模組） 一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參🄵*package*（套件）。

module spec（模組規格） 一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

MRO 請參🄵*method resolution order*（方法解析順序）。

mutable（可變物件） 可變物件可以改變它們的值，但維持它們的 `id()`。另請參🄵*immutable*（不可變物件）。

named tuple（附名元組） 術語「named tuple（附名元組）」是指從 tuple 繼承的任何型🄵或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型🄵或 class 也可以具有其他的特性。

有些🄵建型🄵是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`：

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些 named tuple 是🄵建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

namespace（命名空間） 變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及🄵建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分🄵是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件） 一個 **PEP 420** *package*（套件），它只能作🄵子套件 (subpackage) 的一個容器。命名空間套件可能🄵有實體的表示法，而且具體來🄵它們不像是一個*regular package*（正規套件），因🄵它們🄵🄵有 `__init__.py` 這個檔案。

另請參🄵*module*（模組）。

nested scope（巢狀作用域） 能🄵參照外層定義 (enclosing definition) 中的變數的能力。舉例來🄵，一個函式如果是在另一個函式中被定義，則它便能🄵參照外層函式中的變數。請注意，在預設情🄵下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最🄵層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類🄵） 一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類🄵方法）和 static method（🄵態方法）。

object（物件） 具有狀態（屬性或值）及被定義的行🄵（method）的任何資料。它也是任何*new-style class*（新式類🄵）的最終 base class（基底類🄵）。

package（套件） 一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞🄵的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參🄵*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數） 在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能🄵接受的一個*argument*（引數），或在某些情🄵下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作🄵關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 `/` 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 `*` 來定義的，例如以下的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 `**` 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以🄵一些選擇性的引數指定預設值。

另請參🄵術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差🄵、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目） 在*import path*（匯入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器） 被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參🄵 `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目🄵） 在 `sys.path_hook` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器） 預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件） 一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉🄵🄵 `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分🄵可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP Python Enhancement Proposal（Python 增🄵提案）。PEP 是一份設計🄵明文件，它能🄵 Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成🄵重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計🄵策的記🄵，這些過程的主要機制。PEP 的作者要負責在社群🄵建立共識🄵記🄵反對意見。

請參🄵 **PEP 1**。

portion（部分） 在單一目🄵中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數） 請參🄵*argument*（引數）。

provisional API（暫行 API） 暫行 API 是指，從標准函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示🄵暫行的，理論上🄵不會有重大的變更，但如果核心開發人員認🄵有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更🄵不會無端地🄵生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視🄵「最後的解🄵方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解🄵方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參🄵 **PEP 411** 了解更多細節。

provisional package（暫行套件） 請參🄵*provisional API*（暫行 API）。

Python 3000 Python 3.x 系列版本的🄵稱（很久以前創造的，當時第 3 版的發布是在🄵遠的未來。）也可以縮寫🄵「Py3k」。

Pythonic（Python 風格的） 一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可🄵代物件的所有元素進行🄵圈。許多其他語言🄵🄵有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```
for piece in food:
    print(piece)
```

qualified name（限定名稱） 一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
```

(下页继续)

(繼續上一頁)

```
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，完全限定名稱 *(fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count（參照計數） 對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。參照計數通常在 Python 程式碼中看不到，但它🄵是*CPython* 實作的一個關鍵元素。`sys` 模組定義了一個 `getrefcount()` 函式，程序設計師可以呼叫該函式來回傳一個特定物件的參照計數。

regular package（正規套件） 一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目🄵。

另請參🄵*namespace package*（命名空間套件）。

__slots__ 在 class 🄵部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情🄵。

sequence（序列） 一個*iterable*（可🄵代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，🄵定義了一個 `__len__()` method 來回傳該序列的長度。一些🄵建序列型🄵包括 `list`、`str`、`tuple` 和 `bytes`。請注意，雖然 `dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視🄵對映 (mapping) 而不是序列，因🄵其查找方式是使用任意的*immutable* 鍵，而不是整數。

抽象基底類🄵 (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，🄵不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型🄵，可以使用 `register()` 被明確地🄵🄵。

set comprehension（集合綜合運算） 一種緊密的方法，用來處理一個可🄵代物件中的全部或部分元素，🄵將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會🄵生一個字串 set：`{'r', 'd'}`。請參🄵 comprehensions。

single dispatch（單一調度） *generic function*（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型🄵。

slice（切片） 一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號（subscript notation）"[]"，若要給出多個數字，則在數字之間使用冒號，例如 `in variable_name[1:3:5]`。在括號（下標）符號的🄵部，會使用 `slice` 物件。

special method（特殊方法） 一種會被 Python 自動呼叫的 method，用於對某種型🄵執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底🄵。Special method 在 specialnames 中有詳細🄵明。

statement（陳述式） 陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

text encoding（文字編碼） 將 Unicode 字串編碼🄵位元組的一個編解碼器 (codec)。

text file（文字檔案） 一個能🄵讀取和寫入 `str` 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) 🄵會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開🄵的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參🄵*binary file*（二進制檔案），它是一個能🄵讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

triple-quoted string（三引號F字串） 由三個雙引號 (“) 或單引號 (') 的作F邊界的一個字串。雖然它們FF有提供F於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳F (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫F明字串時特F有用。

type（型F） 一個 Python 物件的型FF定了它是什F類型的物件；每個物件都有一個型F。一個物件的型F可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型FF名） 一個型F的同義詞，透過將型F指定給一個識F符 (identifier) 來建立。

型FF名對於簡化型F提示 *(type hint)* 很有用。例如：

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以寫成這樣，更具有可讀性：

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

請參F `typing` 和 **PEP 484**，有此功能的描述。

type hint（型F提示） 一種*annotation*（F釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型F。

型F提示是選擇性的，而不是被 Python F制的，但它們對F態型F分析工具很有用，F能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型F提示，都可以使用 `typing.get_type_hints()` 來存取。

請參F `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用F行字元） 一種解譯文字流 (text stream) 的方式，會將以下所有的情F識FF一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參F **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數F釋） 一個變數或 class 屬性的*annotation*（F釋）。

F釋變數或 class 屬性時，賦值是選擇性的：

```
class C:
    field: 'annotation'
```

變數F釋通常用於型F提示 *(type hint)*：例如，這個變數預期會取得 `int`（整數）值：

```
count: int = 0
```

變數F釋的語法在 annassign 章節有詳細的解釋。

請參F*function annotation*（函式F釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。

virtual environment（F擬環境） 一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發F套件，而不會對同一個系統上運行的其他 Python 應用程式的行FF生干擾。

另請參F `venv`。

virtual machine（F擬機器） 一部完全由軟體所定義的電腦 (computer)。Python 的F擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之F） Python 設計原則與哲學的列表，其F容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這些🄵明文件

這些🄵明文件是透過 Sphinx（一個專🄵 Python 🄵明文件所撰寫的文件處理器）將使用 reStructuredText 撰寫的原始檔轉🄵而成。

如同 Python 自身，透過自願者的努力下🄵出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，🄵含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者。
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾🄵 Python 這門語言、Python 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl/）的 Guido van Rossum 於 1990 年代早期所創造，目的是作𝔽一種稱𝔽 ABC 語言的後繼者。𝔽管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us/）繼續他在 Python 的工作，𝔽在那𝔽發𝔽了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com 𝔽成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations（現𝔽 Zope Corporation；見 https://www.zope.org/）。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專𝔽擁有 Python 相關的智慧財𝔽權而創立的非營利組織。Zope Corporation 是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參𝔽 https://opensource.org/）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差𝔽。

發𝔽版本	源自	年份	擁有者	GPL 相容性
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備𝔽： GPL 相容𝔽不表示我們是在 GPL 下發𝔽 Python。不像 GPL，所有的 Python 授權都可以讓您發𝔽修改後的版本，但不一定要使您的變更成𝔽開源。GPL 相容的授權使得 Python 可以結合其他在 GPL 下發𝔽的軟體一起使用；但其它的授權則不行。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發🄵成🄵可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和🄵明文件的授權是基於*PSF 授權合約*。

從 Python 3.8.6 開始，🄵明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權合約以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參🄵被收🄵軟體的授權與致謝。

C.2.1 用於 PYTHON 3.9.7 的 PSF 授權合約

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.9.7 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF␣
↪hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative␣
↪works,
   distribute, and otherwise use Python 3.9.7 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.9.7 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.9.7 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made␣
↪to Python
   3.9.7.

4. PSF is making Python 3.9.7 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY␣
↪OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
↪THAT THE
   USE OF PYTHON 3.9.7 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.7
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
```

```
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.7, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material␣
↪breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee. ␣
↪This License
   Agreement does not grant permission to use PSF trademarks or trade name␣
↪in a
   trademark sense to endorse or promote products or services of Licensee,␣
↪or any
   third party.

8. By copying, installing or otherwise using Python 3.9.7, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

BEOPEN PYTHON 開源授權合約第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(下页继续)

(繼續上一頁)

```
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用於 PYTHON 1.6.1 的 CNRI 授權合約

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
```

(下页继续)

(繼續上一頁)

```
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 用於 PYTHON 3.9.7 🄵明文件🄵程式碼的 ZERO-CLAUSE BSD 授權

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 被收🄵軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發🄵版本中所收🄵的第三方軟體。

C.3.1 Mersenne Twister

_random 模組包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載🄵容🄵基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 getaddrinfo() 和 getnameinfo() 函式，它們在 WIDE 專案（http://www.wide.ad.jp/）🄴，於不同的原始檔案中被編碼：

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

asynchat 和 asyncore 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

http.cookies 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追🄵

trace 模組包含以下聲明：

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 模組包含以下聲明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test_epoll` 模組包含以下聲明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

Python/dtoa.c 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉🄴。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 http://www.netlib.org/fp/ 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 hashlib、posix、ssl、crypt 模組會使用它來提升效能。此外，因🄵 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收🄵 OpenSSL 授權的副本：

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
```

(下页继续)

(繼續上一頁)

```
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(下页继续)

(繼續上一頁)

```
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

除非在建置 pyexpat 擴充時設定🄵 `--with-system-expat`，否則該擴充會用一個🄵含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 _ctypes 擴充時設定🄵 `--with-system-libffi`，否則該擴充會用一個🄵含 libffi 原始碼的副本來建置：

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
```

(下页继续)

(繼續上一頁)

```
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 `zlib` 擴充，則該擴充會用一個🄴含 zlib 原始碼的副本來建置：

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 使用的雜🄵表 (hash table) 實作，是以 cfuhash 專案🄵基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
```

(下页继续)

(繼續上一頁)

```
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

除非在建置 `_decimal` 模組時設定🄵 `--with-system-libmpdec`，否則該模組會用一個🄵含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發🄵：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

(下页继续)

(繼續上一頁)

```
* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

版權宣告

Python 和這份🄴明文件的版權:

完整的授權條款資訊請參見沿革與授權。

索引

非字母

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z